Diplomarbeit

Informatik

# Entwurf einer grafischen Entwicklungsumgebung zur Programmierung des LEGO Mindstorms RCX in LeJOS

vorgelegt von

Christian Weddeling
Matr.-Nr.: 3158210

Betreuer:
Prof. Dr. Johannes Magenheim
Michael Dohmen

Bielefeld, 28.03.2007

# Inhaltsverzeichnis


**1 Einleitung** **1**
- 1.1 Motivation . . . 1
- 1.2 Aufgabenbeschreibung . . . 2

**2 Grundlagen** **3**
- 2.1 LEGO MINDSTORMS . . . 3
- 2.2 RCX-Code . . . 6
- 2.3 Robolab . . . 7
  - 2.3.1 Pilot . . . 8
  - 2.3.2 Inventor . . . 9
  - 2.3.3 Investigator . . . 11
  - 2.3.4 Zusammenfassung . . . 11
- 2.4 LeJOS . . . 12
  - 2.4.1 Einschränkungen . . . 12
  - 2.4.2 Programme mit LeJOS übersetzen und übertragen . . . 13
- 2.5 Visuelle Programmiersprachen . . . 14
  - 2.5.1 Definition . . . 15
  - 2.5.2 Sprachkonzepte . . . 16
  - 2.5.3 Ansätze zur Implementierung . . . 18
- 2.6 DEVIL . . . 21
  - 2.6.1 Wie funktioniert DEVIL? . . . 21
  - 2.6.2 Tcl/Tk . . . 26

**3 Entwurf** **29**
- 3.1 Vorarbeiten . . . 29
- 3.2 Prototypen . . . 30
  - 3.2.1 1. Prototyp . . . 30
  - 3.2.2 2. Prototyp . . . 38
  - 3.2.3 Ergebnisse . . . 42
- 3.3 Visuelle Programmierprache . . . 43
  - 3.3.1 Verbindungen . . . 43
  - 3.3.2 Ausgabemöglichkeiten . . . 45
  - 3.3.3 Sensoren . . . 45
  - 3.3.4 Zeit . . . 46
  - 3.3.5 Variablen und Ausdrücke . . . 46
  - 3.3.6 Kontrollstrukturen . . . 48

3.4 Entwurf des Editors . . . . . 49
3.5 Konfiguration . . . . . 50
3.6 Profile . . . . . 52

**4 Implementierung** **55**
4.1 Grammatik und Visualisierung . . . . . 55
4.1.1 Icon . . . . . 55
4.1.2 Ausgänge . . . . . 56
4.1.3 Eingänge . . . . . 58
4.2 Kodegenerierung . . . . . 60
4.3 Kompilierung und Download . . . . . 61
4.4 Konfiguration . . . . . 62
4.5 Profile . . . . . 63

**5 Zusammenfassung und Ausblick** **67**
5.1 Visuelle Programmiersprache . . . . . 67
5.2 Profile und Konfiguration . . . . . 68
5.3 Kodegenerierung und Kompilierung . . . . . 68
5.4 Hardware . . . . . 68

**Selbstständigkeitserklärung** **73**

# Abbildungsverzeichnis


2.1 Der Robotic Command Explorer (RCX) . . . . . 4
2.2 Berührungssensor, Lichtsensor, Rotationssensor, Temperatursensor und Motor . . . . . 4
2.3 Der LEGO Mindstorms Tower . . . . . 5
2.4 RCX code im RIS . . . . . 6
2.5 ROBOLAB im Pilot-Modus . . . . . 8
2.6 ROBOLAB im Inventor-Modus . . . . . 10
2.7 Datenflussdiagramm der Ausdrücke $e = \cos a * \cos b$ und $f = e - (b + c)/d$ 17
2.8 Aufbau eines abstrakten Syntaxbaumes . . . . . 19
2.9 Das Besucher-Muster . . . . . 20

3.1 Formlose Grammatik für den ersten Prototyp, Teil 1 . . . . . 31
3.2 Formlose Grammatik für den ersten Prototyp, Teil 2 . . . . . 33
3.3 Die Grammatik des zweiten Prototypen . . . . . 39
3.4 Die von der Programmiersprache verwendeten Verbindungen . . . . . 44
3.5 Hier sind sämtliche Möglichkeiten zur Ausgabe abgebildet. Die Lampe auf Ausgang B wird am Ende ausgeschaltet, der Motor auf Ausgang A läuft aus. . . . . 45
3.6 Warten auf eine Berührung, Warten auf das Überschreiten eines Schwellwertes beim Lichtsensor. Abfragen des Berührungszustandes und des Lichtes . . . . . 46
3.7 Warten auf Drücken des *Run*-Knopfes auf dem RCX. . . . . 47
3.8 Programm-Schnipsel zur Abfrage des Timers. . . . . 47
3.9 Variablen deklarieren, setzen und abfragen . . . . . 48
3.10 Die Konfiguration . . . . . 50

4.1 Der Konfigurationsdialog für die Profile . . . . . 64

# Tabellenverzeichnis

2.1 Funktionalität der Pilot-Level . . . . . 9
2.2 Funktionalität der Inventor-Level . . . . . 11

3.1 Schnittstelle des Towers nach Plattform . . . . . 51

4.1 Datenstruktur der Konfiguration . . . . . 62

# 1 Einleitung

In diesem Kapitel wird ein kurzer Überblick über den Kontext der Arbeit gegeben und dann die Aufgabenstellung beschrieben. Darüber hinaus wird die Motivation der Arbeit beschrieben.

## 1.1 Motivation

Die Fachgruppe (FG) Didaktik der Informatik (DDI) an der Universität Paderborn verwendet das LEGO Mindstorms System für einen praktischen Zugang zur Informatik. Die mit diesen Baukästen zum Bau von Robotern erstellten, einfachen Programme haben eine große Wirkung und erzeugen bei den Schülern eine hohe Motivation, da schon mit sehr einfachen Programmen gängige Software-Probleme gelöst werden können.

Um den Schülern das Erlernen einer objektorientierten Sprache zu ermöglichen, setzt die FG DDI die Java-Virtual-Machine LeJOS ein. Dadurch können Java-Programme auf den Mindstorms-Robotern ausgeführt werden. Um LeJOS in verschiedenen Entwicklungsumgebungen komfortabel verwenden zu können, wurden die LEGO Mindstorms Tools (LMT) entwickelt und durch Plugins in die Entwicklungsumgebungen integriert, wie zum Beispiel in BlueJ, eine in den nordrhein-westfälischen Klassenzimmern häufig verwendete Entwicklungsumgebung.

Die Schüler müssen ihre Programme auf den LEGO Mindstorms Robotern testen, um ihre Korrektheit festzustellen. Aus Kostengründen können nicht ausreichend viele Roboter zur Verfügung gestellt werden. Viel Unterrichtszeit geht somit durch das Warten auf das Freiwerden des Roboters verloren.

Aus diesem Grund wurde die LEGO Mindstorms Simulationsumgebung (LMS) entwickelt. Mit dieser Umgebung ist es möglich, ein LeJOS-Programm in einer virtuellen Umgebung zu testen. Die Simulation kann in einer dreidimensionalen Visualisierung verfolgt werden. Zwar wird für die Simulation nur ein vereinfachtes Modell der Realität verwendet, jedoch können hierdurch die meisten Fehler der Programme erkannt werden. Somit können die Schüler mit Hilfe des Simulators ihre Programme bis zu dem Reifegrad entwickeln, in dem die Programme keine Flüchtigkeitsfehler mehr haben. Damit wird der Engpass zum Testen an den Robotern entschärft.

Für viele Programmieranfänger ist das Erlernen einer verbalen Programmiersprache nicht sonderlich motivierend, da erst die Syntax einer Programmiersprache erlernt und verstanden werden muss, bevor überhaupt ein funktionsfähiges Programm erstellt werden kann. Somit geht am Anfang viel Zeit verloren, bevor sich die Schüler mit den eigentlichen Problemen und Konzepten der Informatik beschäftigen können.

Dieses Problem lässt sich bis zu einem gewissen Grad durch eine grafische Programmiersprache lösen. Hier muss der Anfänger keine Schlüsselwörter mehr erlernen sondern kann statt dessen gleich fertige Piktogramme verwenden. Auch das aufwändige Eingeben des Programms über die Tastatur kann durch komfortable Zeigeroperationen abgelöst werden. Der Editor der grafischen Programmiersprache kann gleich unterstützend eingreifen und dafür sorgen, dass die Ausdrücke nur an die syntaktisch korrekten Stellen gelangen können und Fehler noch vor der Übersetzung des Programms angezeigt werden.

Für LEGO Mindstorms existiert bereits eine solche grafische Programmiersprache: Robolab. Diese hat jedoch ein paar sprachliche Schwächen und bietet keine Schnittstelle zu Java an. Es ist nicht möglich, ein Robolab-Programm nach Java zu konvertieren noch kann Robolab Java-Programme erzeugen. Somit sind sämtliche Programme verloren, wenn der Schritt zu Java vollzogen wird und es kann auch nicht an Hand von erzeugtem Java-Quelltext beispielhaft erklärt werden, wie ein Java-Programm auszusehen hat.

## 1.2 Aufgabenbeschreibung

Null soll eine grafische Programmiersprache entwickelt werden, die für Programmieranfänger geeignet ist. Dabei soll Robolab als Vorbild dienen, unter anderem um den Umstieg zu erleichtern. Für diese grafische Programmiersprache soll gleichzeitig eine Programmierumgebung entwickelt werden, die den Programmierer bei der Eingabe des Programms unterstützt und ein LeJOS-Programm erzeugt. Es soll ein Zugang zu dem Java-Quellext der erzeugten LeJOS-Programme möglich sein, um diese beim Umstieg auf Java als Beispiel verwenden zu können.

# 2 Grundlagen

In diesem Kapitel werden die für diese Arbeit benötigten Grundlagen vermittelt. Es wird mit der Vorstellung des LEGO MINDSTORMS System begonnen, um dann sich über RCX code und Robolab den Grundlagen der visuellen Programmierung zuzuwenden. Abgeschlossen wird das Kapitel mit einem Überblick über DEVIL.

## 2.1 LEGO MINDSTORMS

LEGO MINDSTORMS ist ein Baukastensystem von LEGO zum Entwickeln von Robotern. Im Prinzip werden dabei die LEGO Technic-Kästen um einen Computer zur Steuerung der Modelle sowie einige Sensoren erweitert.

Der Computer ist der so genannte *Robotic Command Explorer* (siehe Abbildung 2.1 auf Seite 4) oder *RCX* abgekürzt. Dieser extra große LEGO-Stein enthält einen embedded Mikroprozessor und bildet die Grundlage für jeden Roboter. Um mit der Umwelt interagieren zu können, besitzt der Baustein folgende Schnittstellen:

- Drei Anschlüsse für Sensoren
- Drei Anschlüsse für Aktoren
- Eine Infrarotschnittstelle zur Übertragung von Programmen und zur Kommunikation mit anderen RCX.
- Vier Tasten zum Ein- und Ausschalten, Programm auswählen, Programm starten und stoppen und zum Anzeigen von Sensor- oder Aktorwerten.
- Eine LCD-Anzeige zur Text- oder Zahlendarstellung

LEGO stellt verschiedene Sensortypen zur Verfügung (siehe Abbildung 2.2 auf Seite 4). Im *Robotic Invention Set* sind zum Beispiel Berührungssensoren und Lichtsensoren enthalten. Des weiteren sind noch Temperatur- und Rotationssensoren erhältlich. Als Aktoren finden verschiedene, von LEGO hergestellte Elektromotoren Verwendung.

Es gibt drei unterschiedliche Versionen der RCX-Bausteine: 1.0, 1.5, 2.0. Diese unterscheiden sich minimal in der Hardware und der Firmware. Die Unterschiede in der Hardware sind nicht funktional, das heißt, es kann jeder Sensor und Aktor an jeden RCX-Baustein angeschlossen werden. Jedoch ist die Infrarotübertragung zu einem RCX der Version 1.0 nur mit einfacher Geschwindigkeit möglich, während die anderen Bausteine auch die vierfache Geschwindigkeit unterstützen (siehe Kapitel 2.4

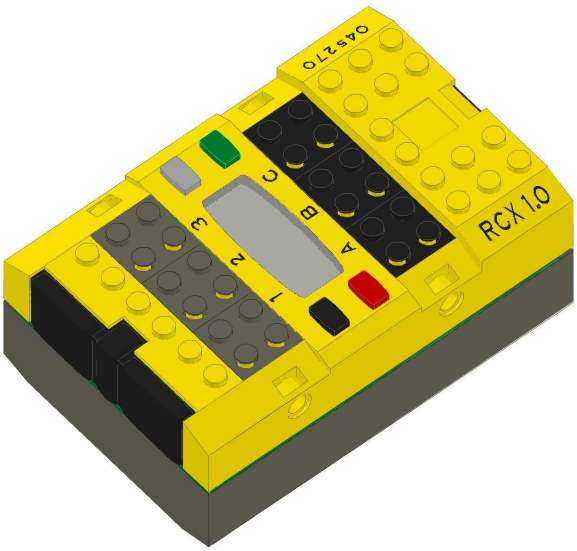


Abbildung 2.1: Der Robotic Command Explorer (RCX)

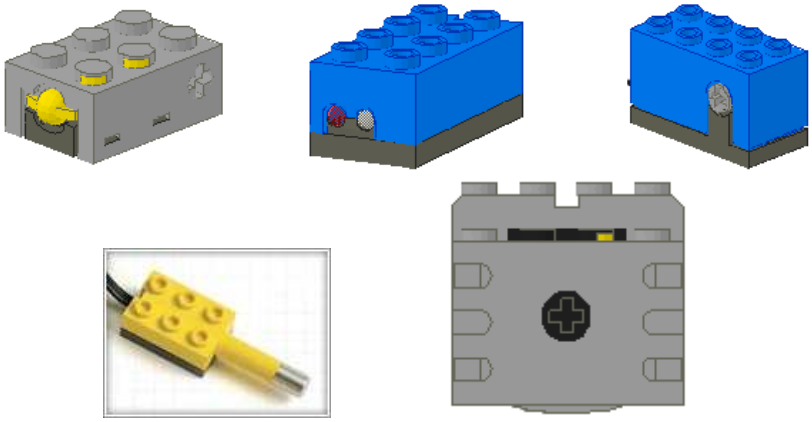

Abbildung 2.2: Berührungssensor, Lichtsensor, Rotationssensor, Temperatursensor und Motor

Abbildung 2.3: Der LEGO Mindstorms Tower

auf Seite 12). Für jede RCX-Version gibt es eine entsprechende Firmware-Version. Hier gibt es Unterschiede, die jedoch von LeJOS verdeckt werden. In dieser Arbeit kann somit jede Version des RCX-Bausteins verwendet werden. Dem Autor stand ein Baustein der Version 1.0 zur Verfügung.

Über die Infrarotschnittstelle werden Steuerprogramme übertragen, die die Sensoren auswerten und dementsprechend die Aktoren steuern können. Die Übertragung vom PC zum RCX erfolgt mit dem Tower (siehe Abbildung 2.3 auf Seite 5), der bei den älteren Geräten über die serielle Schnittstelle angeschlossen wird. Die neueren Tower verfügen über eine USB-Schnittstelle.

Zur Erstellung der Steuerprogramme stellt LEGO zwei Entwicklungsumgebungen zur Verfügung. Diese heißen *RCX code* (siehe Kapitel 2.2 auf Seite 6) und *ROBOLAB* (siehe Kapitel 2.3 auf Seite 7). Daneben sind noch eine Reihe inoffizieller Entwicklungssysteme entstanden wie NQC, BrickOS (ehemals LegOS), pbForth und LeJOS.

NQC ist eine C ähnliche Sprache und verwendet als einziges Entwicklungssystem die original Firmware von LEGO. Dies hat den Vorteil, dass gleichzeitig NQC-, RCX-Code- und ROBOLAB-Programme auf dem RCX sein können. Die Firmware verwendet einen Byte-Code-Interpreter um den vorhandenen Platz effizient nutzen zu können, also werden die Programme von NQC, RCX-Code und ROBOLAB interpretiert.

Abbildung 2.4: RCX code im RIS

Die anderen Umgebungen bringen jeweils ihre eigene Firmware mit, die zu Beginn aufgespielt werden muss. BrickOS basiert auf dem gcc mit einem Hitachi-H8 Backend, also einem Cross-Compiler. Es werden die Programmiersprachen C und C++ unterstützt. LeJOS basiert auf Java und stellt eine Java Virtual Machine für den RCX zur Verfügung. Weitere Erklärungen zu LeJOS finden sich in Kapitel 2.4 auf Seite 12.

## 2.2 RCX-Code

Beim *Robotic Invention Set* (*RIS*) wird auch eine Programmierumgebung mitgeliefert die eigentlich *RCX code* heißt. Allerdings wird RCX code meistens mit dem Namen des Baukastens bezeichnet, nämlich RIS.

RCX-Code ist für Programmieranfänger entwickelt worden. Deshalb lassen sich die Sprachelemente nicht frei positionieren, sondern der Editor sorgt für die korrekte Anordnung. Auch die Verbindung der einzelnen Elemente wie beim Inventor entfällt. Man zieht ein Element oder eine ganze Kette von Elementen einfach unter ein anderes Element und schon wird automatisch eine Verbindung hergestellt. Auf Kopieren, Ausschneiden und Einfügen kann verzichtet werden, da ein Element mit der Maus irgendwo auf die Arbeitsfläche gezogen wird und dort zwischengelagert wird. Die Mehrfachselektion erfolgt entweder wie gewohnt über das Gummiband oder indem ein Element angeklickt wird und alle Elemente bis zum Programmende automatisch mit ausgewählt werden.

Die Sprache unterstützt alle bekannten Aktoren und Sensoren, Timer, Verzweigungen, Schleifen, Funktionen und globale Variablen. Ereignisse und Threads sind vermischt, das heißt die Behandlung eines Ereignisses verwendet einen eigenen Thread.

Eine Kommunikation zwischen den Threads ist nur über die Variablen möglich. Es gibt sogenannte *Building Blocks*, das sind vom Benutzer erzeugbare Sprachschnipsel, die wie eine eigenständige Anweisung in das Programm eingefügt werden. Die Building Blocks sind Prozeduren ohne Parameter und Rückgabewerten ähnlich. Sie haben jedoch einen interessanten Unterschied: es gibt keinen Unterschied zwischen Definition und Aufruf einer Prozedur. Dies ermöglicht dem Benutzer, an jeder Benutzung eines Building Blocks die enthaltenen Anweisungen einzusehen und diese zu ändern. Diese Änderung wirkt sich dann sofort auf sämtliche Benutzungen dieses Building Blocks aus.

Eine sehr schöne Eigenschaft ist die Möglichkeit Blöcke, wie bei Building Blocks, Schleifen und Verzweigungen zu falten, wodurch sich das Programm ein wenig übersichtlicher gestaltet. Auch lässt sich die Anzeige in Grenzen vergrößern und verkleinern, wodurch ein größerer Überblick gewährleistet ist. Jedoch ist die Schrift der Anweisungen bei einer Verkleinerung kaum noch lesbar.

Bei RCX code werden die Anweisungen wie bei einer imperativen Sprache aneinander gereiht. Durch entsprechende Kontrollstrukturen werden Schleifen und Bedingungen realisiert. Die Anweisungen können nicht beliebig auf dem Bildschirm platziert werden, sondern werden an vom Computer berechneten Stellen eingefügt. Dieses Verfahren senkt die Fehlermöglichkeiten, da es verhindert, ein fehlerhaftes Programm herzustellen. So erstellte Diagramme lassen sich immer in ein funktionsfähiges Programm übersetzen. Ob das Programm auch die Problemstellung erfüllt, ist durch diese Editorunterstüzung nicht sichergestellt.

Wie in Abbildung 2.4 zu sehen ist werden keine Piktogramme zur Darstellung von Anweisungen und Kontrollstrukturen verwendet. Stattdessen wird die Funktion einer Anweisung durch einen kurzen Text wie zum Beispiel *Repeat* oder *Set Power AC 8* beschrieben. Die Anweisungen werden durch grüne Rechtecke dargestellt, die Schleifen durch orange und die Bedingungen durch violette. Die Parameter werden über Dialoge eingestellt.

## 2.3 Robolab

ROBOLAB baut auf dem Programm LabVIEW der Firma National Instruments auf. Dieses Programm wurde ursprünglich zur Erfassung und Anzeige von Daten und zur Steuerung von Anlagen entwickelt und wird in Wissenschaft, Forschung und Industrie eingesetzt.

ROBOLAB ist in drei Bereiche aufgeteilt: Pilot, Inventor und Investigator. Pilot ist auf absolute Programmieranfänger zugeschnitten und macht es nahezu unmöglich, ein nicht funktionsfähiges Programm zu erstellen. Der Inventor ist an den schon etwas fortgeschrittenen Programmierer gerichtet und ermöglicht wesentlich komplexere Programme. Investigator ist der Bereich zum Sammeln und Auswerten von Daten. Darüberhinaus sind die Bereiche Pilot und Inventor noch in Level unterteilt. In den niedrigeren Leveln stehen nur die einfacheren Programmelemente zur Verfügung wogegen die Möglichkeiten mit zunehmendem Level immer komplexer

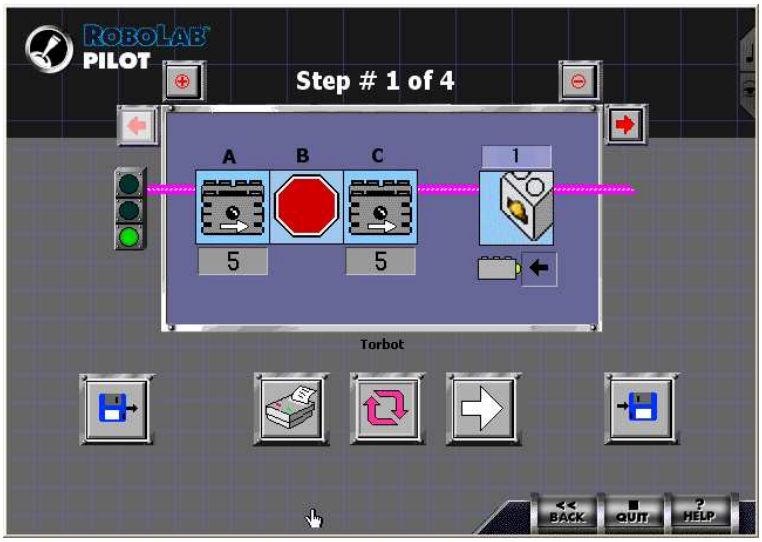


Abbildung 2.5: ROBOLAB im Pilot-Modus

werden.

### 2.3.1 Pilot

Wie bereits ausgeführt ist der Pilot für Programmieranfänger gedacht und so aufgebaut, dass nur funktionstüchtige Programme erstellt werden können. Pilot-Programme sind demnach eigentlich fertige Programme, die abgeändert werden. In Level 1 besteht dieses Programm aus einem Befehl zum Setzen des Ausgangs A und einem Befehl zum Warten. Der Programmierer kann jetzt entscheiden, in welche Richtung ein eventuell vorhandener Motor am Ausgang A drehen soll oder ob dort eine Lampe angeschaltet wird. Anschließend kann festgelegt werden, wie lange der Ausgang in diesem Zustand bleiben soll, in dem die angegebene Zeit lang gewartet wird. Beim Beenden des Programms werden alle Ausgänge wieder ausgeschaltet. In diesem Level gibt es demnach nicht viele Möglichkeiten, es wird jedoch sehr schnell ein funktionierendes Ergebnis erzielt und die weiteren Funktionen der Oberfläche werden auf das nötigste reduziert. Man kann das Programm nur zum RCX übertragen oder ausdrucken.

In Abbildung 2.5 ist der höchste Level zu sehen. In der Mitte wird ein Schritt des Programms dargestellt, auf dem links eine Gruppe von drei Piktogrammen für die Steuerung der drei Ausgänge vorgesehen ist. Rechts befindet sich ein Piktogramm, dass das Warten auf ein Ereignis von einem Eingang oder einfach das Warten für einen gewissen Zeitraum ermöglicht. Es besteht die Möglichkeit, weitere solche Schritte durch die + und - Knöpfe oberhalb des Blattes hinzu zu fügen oder zu entfernen. Hierdurch kann das Programm auf die benötigte Größe angepasst werden. Durch die Benutzung der *Pfeil*-Knöpfe links und rechts vom Schritt kann im Programm geblättert werden. Unter dem Schritt herum befinden sich Knöpfe

| Funktionalität | Level 1 | Level 2 | Level 3 | Level 4 |
|---|---|---|---|---|
| Laden | - | - | + | + |
| Speichern | - | - | + | + |
| Drucken | + | + | + | + |
| Programm übertragen | + | + | + | + |
| Endlosschleife | - | - | + | + |
| Ausgänge | 1 | 2 | 3 | 3 |
| Anzahl der Schritte | 1 | 1 | 2 in 1 | beliebig |
| Schritte Blättern | - | - | - | + |
| Motor vorwärts | ohne Geschwindigkeit | + | + | + |
| Motor rückwärts | ohne Geschwindigkeit | + | + | + |
| Lampe | ohne Geschwindigkeit | + | + | + |
| Stop | - | + | + | + |
| Warten | + | + | + | nur mit Parameter |
| Berührungssensor | - | + | + | mit Parameter |
| Lichtsensor | - | - | + | mit Parameter |

Tabelle 2.1: Funktionalität der Pilot-Level

für das Laden, Speichern, Drucken des Programms sowie noch zum Übertragen des Programms zum RCX.

Ein weiterer Knopf unterhalb des Blattes legt fest, ob das Programm nur einmal oder in einer Endlosschleife durchlaufen werden soll. Dies kann die im Pilot fehlende Schleife begrenzt ersetzen.

### 2.3.2 Inventor

Beim Inventor dagegen müssen die einzelnen Befehle selbst verbunden werden. Es wird zwar vor der Übertragung zum RCX ein Plausibilitätstest durchgeführt, jedoch wäre es trotzdem möglich, ein funktionsuntüchtiges Programm zu erstellen. Im Level 1 erhält man als Vorlage ein Beispiel das Level 2 des Pilot entspricht. Trotzdem erscheint das Programm identisch. Die Gruppierung in Befehle zur Ausgabe und zum Warten entfällt. Die Ausgänge müssen vor dem Beenden des Programms zurückgesetzt werden, da sie sonst immer eingeschaltet bleiben. Ein fahrender Roboter würde so zum Beispiel immer weiter fahren selbst wenn das Programm beendet wurde. Der angebotene Befehlsumfang entspricht ungefähr dem des Pilot Level 2.

Die Vorlage ist zwar ein guter Anhaltspunkt, jedoch können nach Belieben Befehle hinzugefügt und entfernt werden. Es besteht keine Bindung mehr an die Positionsvorgaben des Pilot, und die Befehle können an jede beliebige Stelle verschoben werden. Die freie Positionierbarkeit kann zum Bilden von Gruppen genutzt werden. Um den Überblick zu behalten, muss jedoch immer die Positionierung vorgenommen werden. Von Beginn an können Programme geladen und gespeichert werden, wobei jedoch die Funktionalität in den Menüs und nicht wie beim Pilot auf extra Knöpfen

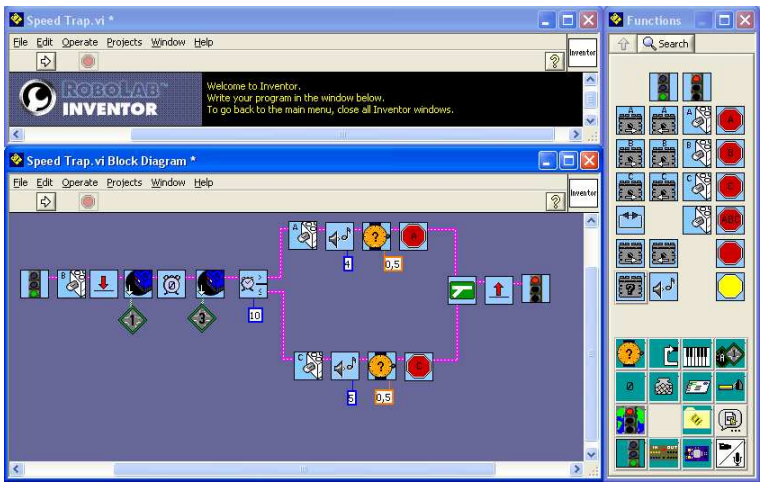
Speed Trap.vi *
File Edit Operate Projects Window Help
ROBOLAB INVENTOR
Welcome to Inventor.
Write your program in the window below.
To go back to the main menu, close all Inventor windows.
Speed Trap.vi Block Diagram *
File Edit Operate Projects Window Help
Functions
Search

Abbildung 2.6: ROBOLAB im Inventor-Modus

zu finden ist und auch nicht der *Level-Verwaltung* unterliegt.

Während des Editerens wird immer überprüft, ob der derzeitige Zustand des Programms syntaktisch korreckt ist. Diese Überprüfung ist nicht perfekt. Wenn vergessen wird, eine Schleife zu schließen, dann wird das von der Syntaxprüfung nicht erkannt. ROBOLAB nimmt dann an, dass die Schleife bis zum Ende des Programms geht und das Programm lässt sich zum RCX übertragen. So ein Programm scheint den RCX in einen undefinierten Zustand zu versetzen, da nach der ersten Ausführung des defekten Programms das Herunterladen eines korrekten oder anderen Programms nicht mehr möglich war. Es wird immer das defekte Programm ausgeführt war. Erst nach einmaligen Aus- und Anschalten verhält sich der RCX wieder normal.

Die Variablen sind in ROBOLAB immer vorhanden. Intern werden sie als ein Feld von 16-Bit-Zahlen gehalten. Die Variablen werden beim Programmstart nicht zurückgesetzt, deshalb muss der Programmierer darauf achten, dass alle benutzten Variablen immer im Programm vor deren Verwendung initialisiert werden. Ansonsten wird das Verhalten der Programme unvorhersehbar.

Wie beim Pilot sind auch beim Inventor vier verschieden Level vorhanden. Diese wirken sich im Gegensatz zum Pilot nicht auf den Editor aus. Das heißt, dass das Laden, Speichern, Drucken und die Größe des Programms immer unbeschränkt sind. Die Level wirken sich nur auf die visuelle Programmiersprache aus. Eine Übersicht der Funktionalität finden sie in Tabelle 2.2 auf Seite tab:inventorLevel. Um den Umfang der Tabelle auf eine erträgliche Größe zu beschränken, wurde nicht wie beim Pilot (siehe Tabelle 2.1 auf Seite 2.1 jedes einzelne Sprachelement aufgeführt. Stattdessen wurden die Sprachelemente in Gruppen zusammengefasst und die Anzahl der Piktogramme in die Tabelle eingetragen.

| Funktionalität | Level 1 | Level 2 | Level 3 | Level 4 |
|---|---|---|---|---|
| Ausgaben | 14 | 19 | 19 | 30 |
| Warten auf... | 8 | 12 | 12 | 80 |
| Kontrollstrukturen | - | 2 | 16 | 195 |
| Modifizierer | - | 12 | 13 | 80 |
| Musik | - | - | 20 | 25 |
| Verhalten | - | - | - | 13 |
| Multimedia | - | - | - | 27 |
| Investigator | - | - | - | 41 |
| Zurücksetzen | - | - | - | 21 |
| Variablen | - | - | - | 48 |
| Kommunikation | - | - | - | 18 |
| NXT | - | - | - | + |

Tabelle 2.2: Funktionalität der Inventor-Level

### 2.3.3 Investigator

Der Investigator dient zur Datenerfassung und zur Auswertung dieser Daten. Dieses Aufgabengebiet wurde von der Aufgabenstellung ausgeklammert und wird hier nur kurz betrachtet. Das Aufgabengebiet umfasst die Teilbereiche Datenerfassung und Auswertung. Der Teilbereich Datenerfassung verhält sich fast so wie der Investigator, nur gibt es zusätzlich Befehle zum Sammeln von Daten. Diese werden dann nach der Erfassung zum Computer übertragen und können in Diagrammen ausgewertet werden.

### 2.3.4 Zusammenfassung

Zusammenfassend kann festgestellt werden, dass ROBOLAB ein gelungenes Programm ist, welches sowohl für den Anfänger als auch für den Profi geeignet ist. Überzeugend ist das Konzept des Pilots für Neulinge, da es zwar nur einen begrenzten Lösungsraum zulässt, dafür aber einen schnellen Einstieg in einfache Programme ermöglicht. Man gewöhnt sich an die verwendeten Piktogramme und erhält somit vorab die Grundlagen für den Inventor. Die Eigenschaften der Befehle wie Ports und Ausgangsleistung oder die zu wartende Zeit stehen immer direkt bei den Piktogrammen wodurch die Werte und auch die Defaultwerte immer gut sichtbar sind. Die Defaultwerte sind beim Inventor auf Grund des dann nicht vorhandenen Modifizierers nicht sichtbar. Die Funktion eines Piktogramms lässt sich herausfinden indem mit Kontextmenü eine Kurzhilfe aufgerufen wird.

Der Inventor bietet das Potential zur Lösung sämtlicher Aufgabenstellungen, die sich mit einem RCX ergeben. Er bietet eine vollständige Sprache, wobei die Bedienung schon etwas Grundwissen erfordert. Bei der großen Anzahl Piktogramme wäre eine textuelle Beschreibung wünschenswert die zum Beispiel als Tooltip erscheint

wenn der Zeiger über dem Piktogramm verweilt.

## 2.4 LeJOS

LeJOS ist eine virtuelle Maschine für Java-Programme, die für den RCX entwickelt wurde. LeJOS bietet die Vorteile einer objektorientierten Sprache mit preemptiven Threads, multidimensionalen Feldern, Rekursion und Ausnahmen. Es werden Fließkomma-Operationen unterstützt, wenn auch nur bis 32 Bit Länge. Neben den Grundrechenarten gibt es noch die Klasse `java.lang.Math` mit einer Vielzahl weiterer mathematischer Funktionen und Konstanten. Es können auch Textkonstanten verwendet werden.

### 2.4.1 Einschränkungen

Es gibt jedoch auch einige Einschränkungen, was bei einer Virtual Machine mit einer Größe von circa 16 Kb nicht verwundert. Die wichtigste Einschränkung ist die, dass es keine Garbage Collection gibt. Dies ist für eine Sprache wie Java schon eine sehr starke Einschränkung, da alle Objekte, die mit new erzeugt werden, nicht mehr aus dem Speicher entfernt werden. Dadurch fallen viele objektorientierte Programmiertechniken weg.

Um sicher zu stellen, dass keine Einweg-Objekte erzeugt werden, deklariert der Programmierer die Variablen als statisch. Sollte dies nicht ausreichen, können auch Mehrweg-Objekte erzeugt werden. Hierzu deklariert der Programmierer ein statisches Feld von Objekten in ausreichender Größe. Diese Objekte werden dann immer wieder verwendet.

Auf Grund der fehlenden Garbage Collection geht auch die `String`-Klasse verloren. Ein `String` ist in Java konstant, also muss für jede Änderung ein neues `String`-Objekt erzeugt werden. Allerdings ist der RCX für die Verwendung von Strings nicht sonderlich geeignet, da ihm ein passendes Ausgabegerät für die Anzeige von Texten fehlt und somit der Verlust zu verschmerzen ist. Alle primitiven Typen wie `int`, `long`, `float` und `double` werden jedoch auf dem Stack erzeugt und stehen somit nicht unter der Verwaltung der Garbage Collection. Sie können weiterhin wie gewohnt eingesetzt werden, wie auch in der Mail von (Stuber und Denham, 2006) nachzulesen ist.

Die primitiven Typen haben jedoch einige Einschränkungen. Grundsätzlich kann gesagt werden, dass nur Datentypen bis 32 Bit Länge komplett implementiert sind. Der Typ `long` kann zwar ganzzahlige 64-Bit-Zahlen speichern, jedoch kann damit nicht gerechnet werden. Für Berechnungen muss ein `long` in ein `int` umgewandelt werden. Dabei ist auf die unterschiedliche Größe der Zahlenräume zu achten. Der Typ `double` wird automatisch auf 32 Bit gekürzt. Durch die Kürzung wird die Mantisse von 53 auf 24 Bits verkleinert und es geht Genauigkeit verloren. Auch der Zahlenraum wird kleiner, da der Exponent von 11 auf 8 Bits gekürzt wird, was gegebenenfalls zu Fehlern führt.

Die Beschränkung der primitiven Zahlentypen auf 32 Bit ist jedoch nicht relevant, da Berechnungen auf einem RCX, für die Zahlentypen mit 64 Bit Länge notwendig sind, nicht ausgeführt werden. Dafür ist der RCX weder gedacht noch gebaut worden. Das Vorhandensein von `float` ist zu begrüßen, da dann trigometrische Berechnungen möglich sind.

Felder sind in LeJOS auf 511 Elemente beschränkt. Eine Einschränkung der Feldgröße ist klar dem vorhandenen Speicherplatz im RCX geschuldet, aber die Festsetzung auf 511 Elemente konnte nicht geklärt werden.

Ansonsten sind `Switch`-Statements nicht gestattet. Diese lassen sich jedoch ohne weiteres mit `If`-Statements nachbilden. Es fehlen auch verschiedene Mechanismen im Zusammenhang mit der Polymorphie von Objekten, welche jedoch in dieser Arbeit nicht verwendet werden.

Natürlich fehlen auf Grund der geringen Speichermenge auch die meisten APIs des JDK, aber die dazu benötigten Aus- und Eingabegeräte sind nicht vorhanden. Jedoch sind Threads und Ereignisse vorhanden. Die Klasse Thread implementiert alle notwendigen Methoden, die für die Benutzung wichtig sind. Die Ereignisse sind über Rückrufmethoden implementiert, die jedoch zur Datenübergabe keine Klassen sondern ausschließlich primitive Datentypen verwenden, was der fehlenden Garbage Collection geschuldet ist. Ansonsten werden Events unter LeJOS genauso verwendet wie im normalen Java.

Weitere Eigenschaften der virtuellen Maschine können in dem *README* (LeJOS, 2002) der Distribution nachgelesen werden.

Zusammanfassend lässt sich sagen, dass die objektorientierten Fähigkeiten auf Grund des fehlenden Garbage Collectors weitestgehend verloren gehen und somit nur der imperative Teil der Sprache genutzt werden kann. Jedoch stellt sich auch die Frage, wie viel Objektorientierung mit 32 Kb Speicher realisiert werden kann und wie viel Objektorientierung in den kleinen RCX-Programmen überhaupt sinnvoll möglich ist.

### 2.4.2 Programme mit LeJOS übersetzen und übertragen

Die zu erstellende Entwicklungsumgebung soll Programme mit LeJOS übersetzen und zum RCX übertragen. Deshalb soll dieser Prozess genauer betrachtet werden. Standardmäßig sieht dieser Prozess folgende Schritte vor:

1. Falls noch nicht geschehen, die Firmware herunterladen.
2. Den Java-Quelltext übersetzen.
3. Die einzelnen `.class`-Dateien zu einer Datei zusammenfassen.
4. Die zusammengefasste Datei zum RCX übertragen.

Für jede dieser Aufgaben stellt LeJOS ein Programm zur Verfügung. Dabei gibt es von jedem Tool zwei Varianten: die Java-Tools und die C-Tools. Der Hauptunterschied zwischen beiden Varianten ist die Programmiersprache, in der die Programme

implementiert worden sind. Die Java-Tools haben den Vorteil, dass sie auf jedem Betriebssystem ohne neuerliche Übersetzung laufen. Die C-Tools hingegen werden nur für Windows übersetzt mitgeliefert, alle anderen Plattformen benötigen eine neuerliche Übersetzung.

Die Java-Tools benötigen noch ein paar Einstellungen, damit sie funktionieren. Sie benötigen auf jeden Fall eine Java-Virtual-Machine, die im Suchpfad liegen sollte, genauso wie den Java-Compiler, der für das Übersetzen benötigt wird. Dann benötigen die Java-Tools noch die Umgebungsvariable `LEJOS_HOME`, die auf das LeJOS-Verzeichnis zeigt. Abschließend muss den Werkzeugen noch mitgeteilt werden, wo der Tower angeschlossen ist. Dazu wird die Umgebungsvariable `RCXTTY` passend gesetzt.

Nun kann mit dem Skript `firmdl.bat` die Firmware auf den RCX übertragen werden. Dies muss als erstes gemacht werden, bevor ein LeJOS-Programm auf den RCX übertragen werden kann. Die LeJOS-Firmware enthält die Virtual-Machine.

Ein Quelltext wird mit `lejosjc.bat` übersetzt. Dabei müssen je nach verwendetem JDK noch zusätzliche Parameter angegeben werden. Ab JDK 1.2 muss die Option `-target 1.1` gesetzt werden, da die LeJOS-Virtual-Machine nur Byte-Code der Version 1.1 verarbeiten kann. Ab JDK-Version 1.4 muss die Option `-source 1.3` gesetzt werden, damit die neuen Sprachkonzepte nicht verwendet werden, wie zum Beispiel die *Generics*. Die zu übersetzenden Java-Dateien sowie weitere Bibliotheken sind wie gewohnt anzugeben.

Nach dem Übersetzen muss das Programm mit `lejoslink.bat` gelinkt werden. Im Gegensatz zu normalen Java-Programmen ist das notwendig, um das Vorhandensein aller benötigten Klassen auf dem RCX sicherzustellen. Dabei werden aus den Bibliotheken nur die benötigten Klassen verwendet um Speicherplatz zu sparen. Dieses Werkzeug überträgt das gelinkte Programm gleich zum RCX, welches durch die Option `-o DATEI` in die angegebene Datei geschrieben werden kann. Dann wäre eine Übertragung zum RCX durch das Skript `lejosdl.bat` möglich.

## 2.5 Visuelle Programmiersprachen

Da wir uns in dieser Arbeit mit visuellen Programmiersprachen beschäftigen, möchte ich kurz den Begriff *visuelle Programmiersprache* definieren, da dieser Begriff noch nicht Eingang in die gängigen Lexikas gefunden hat beziehungsweise oft mit den Komponenten einer Programmiersprache verwechselt wird, die für die Oberflächendarstellung zuständig ist.

Im zweiten Teil wollte ich kurz die möglichen Sprachkonzepte vorstellen, wovon die zwei für diese Arbeit von weiteren Interesse sind. Abschließend werde ich noch kurz auf Ansätze zur Implementierung einer visuellen Programmiersprache eingehen.

### 2.5.1 Definition

Leider ist der Begriff *visuelle Programmiersprache* nicht eindeutig definiert. Die erste Anlaufstelle, das Wikipedia, führt diesen Begriff gar nicht als eigenständigen Begriff (Visuelle Programmiersprache (2007)). Statt dessen findet man die *visuelle Programmierung* unter dem Begriff *Programmiersprache* als Verweis auf *visuelle Programmierumgebung*.

Mit dieser Definition ließ sich nicht weiter arbeiten, somit wurde sich an den Definitionen aus Schiffer (1998) orientiert. Obwohl dieses Buch mit einer Drucklegung in 1998 schon in die Jahre gekommen ist, lässt es sich hervorragend als Einstieg in die Materie verwenden. Um hier die notwendigen Grundlagen darzulegen wurde das Buch von Schiffer als Grundlage verwendet. Deshalb sind die sinngemäß wiedergegebenen oder wörtlich zitierten Passagen nicht weiter kenntlich gemacht worden.

Schiffer definiert den Begriff *visuelle Programmiersprache* wie folgt:

> Eine visuelle Programmiersprache ist eine visuelle Sprache zur vollständigen Beschreibung der Eigenschaften von Software. Sie ist entweder eine Universalprogrammiersprache oder eine Spezialprogrammiersprache.

Einige Begriffe dieser Definition benötigen weitergehender Erläuterungen. Mit einer *Universalprogrammiersprache* kann jeder Algorithmus beschrieben werden, der auf einer Turingmaschine lösbar ist. Da alle berechenbaren Probleme auf einer Turingmaschine lösbar sind, werden Universalsprachen als vollständig oder berechnungsuniversell bezeichnet.

Eine *Spezialprogrammiersprache* kann dagegen nicht alle berechenbaren Probleme lösen, da ihr wichtige Sprachkonstrukte wie zum Beispiel Schleifen fehlen. Jedoch kann mit einer Spezialsprache ein vollständiges Programm definiert werden.

Nun stellt sich die Frage, was eine *visuelle Sprache* ist. Schiffer definiert sie so:

> Eine visuelle Sprache ist eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung.

*Statische Zeichengebung* entspricht dem Niederschreiben der Sprache auf Papier oder dem Ablegen des Programms in einer Datei, also *bleibende Zeichengebung*. *Dynamische Zeichengebung* ist demnach die *einmalige Zeichengebung* wie etwa die Gebärdensprache oder das Vorzeigen einer Aktionsfolge zur Generierung eines Programms.

Eine *formale Sprache* besteht aus einem Alphabet, der Syntax zur Bildung von Ausdrücken aus den Elementen des Alphabets und einer Semantik zur Ermittlung der Bedeutung von Ausdrücken. Damit ein Ausdruck gültig ist, muss er in einer formalen Sprache der Syntax genügen. Ein Ausdruck ergibt dann einen Sinn, wenn er der Semantik entspricht. Semantik und Syntax müssen selbst nicht in einer formalen Sprache vorliegen, jedoch müssen sie vollständig und widerspruchsfrei beschrieben sein.

Zu erklären bleibt was das syntaktische und was das semantische Konstrukt einer visuellen Sprache ist. So ist ein Pfeil zwischen zwei Elementen einer Grafik ein *visuelles syntaktisches Konstrukt* und die Farbe des Pfeils ein *visuelles semantisches Konstrukt*.

Der Begriff *visuell* wird von Schiffer so definiert:

> Visuell ist die Bezeichnung für jene Eigenschaft eines Objekts, durch die mindestens eine Information über das Objekt, die für das Erreichen eines Handlungsziels unverzichtbar ist, nur durch das visuelle Wahrnehmungssystem des Menschen gewonnen werden kann.

Ein *Objekt* ist ein logischer Gegenstand. Im Sinne der Programmierung ist ein Objekt eine Funktion, Anweisung, Schleife oder ähnliches.

Zur Abgrenzung gegenüber visuellen Programmiersprachen soll der Begriff der *verbalen Programmiersprache* näher beschrieben werden. Bei einer verbalen Programmiersprache sind die visuellen Elemente syntaktisch und semantisch bedeutungslos. So könnte jeder beliebige, in Java implementierte Algorithmus in einer Zeile dargestellt werden. Obwohl so ein Programm für einen Menschen kaum noch lesbar ist, kann es vom Compiler übersetzt werden. Es kann auch vorgelesen werden.

### 2.5.2 Sprachkonzepte

Schiffer (1998) hat auf den Seiten 131 bis 169 die Sprachkonzepte der visuellen Programmiersysteme beschrieben. Es werden folgende Konzepte behandelt:

- Steuerflussorientierte visuelle Programmiersysteme
- Datenflussorientierte visuelle Programmiersysteme
- Funktionsorientierte visuelle Programmiersysteme
- Objektorientierte visuelle Programmiersysteme
- Constraintorientierte visuelle Programmiersysteme
- Regeloriorientierte visuelle Programmiersysteme
- Beispielorientierte visuelle Programmiersysteme
- Formularorientierte visuelle Programmiersysteme
- Multiparadigmenorientierte visuelle Programmiersysteme

Nicht alle hier aufgeführten Konzepte sind für uns interessant. Für die meisten Konzepte gibt es keine visuelle Programmierumgebung mit der Programme für einen RCX erstellt werden können. Im Endeffekt bleiben die steuerflussorientierten und objektorientierten Konzepte übrig.

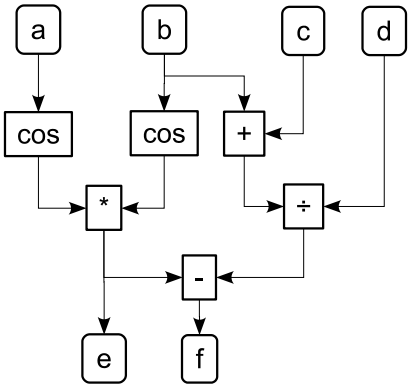


Abbildung 2.7: Datenflussdiagramm der Ausdrücke $e = \cos a * \cos b$ und $f = e - (b + c)/d$

Das Steuerflusskonzept hat seinen Namen durch Auslösung der Befehlsaktivierung durch die Programmsteuerung. Der Steuerfluss kann durch Anweisungssequenzen, Komponentennetze und Transitionsnetze festgelegt werden.

**Anweisungssequenzen** sind eine Folge von Operationen, die in der Reihenfolge der Niederschrift ausgeführt werden. Um die lineare Ausführung zu modifizieren gibt es Anweisungen zur Ablaufsteuerung wie zum Beispiel Schleifen, Verzweigungen und Funktions- und Prozeduraufrufe. Diese Art der Programmierung entspricht der imperativer Programmiersprachen.

**Komponentennetze** sind durch Kanäle verbundene Sofwarekomponenten. Ein Kanal verbindet einen Ausgang mit einem oder mehreren Eingängen. Über diese Kanäle reisen Token von dem Ausgang einer Komponente zu einem Eingang einer anderen Komponente und aktivieren diese Komponente. Ein Komponentennetz mit genau definierten Ein- und Ausgängen ergibt wieder eine neue Komponente.

**Transitionsnetze** beschreiben zum Beispiel Automaten und Petrinetze. Der Vorteil von Transitionsnetzen ist die Möglichkeit des mathematischen Beweises auf korrektes Systemverhalten.

Das objektorientierte Konzept ist in der Welt der verbalen Programmiersprachen weit verbreitet. Bei den visuellen Programmiersprachen konnte es sich bislang noch nicht durchsetzen. UML wäre vielleicht ein Kandidat, jedoch scheinen die dynamischen Ansichten von UML nur ein *prettyprint* von einer verbalen Programmiersprache zu sein.

Datenflussorientierte und funktionsorientierte visuelle Programmiersprachen sind miteinander verwandt, da sie auf den selben theoretischen Grundlagen aufbauen. Datenflussorientierte Programmiersprachen unterscheiden sich wesentlich von steuerflussorientierten Programmiersprachen. Während bei einer steuerflussorientierten Programmiersprache ein Programmzähler die Abfolge der einzelnen Anweisungen

kontrolliert und Variablen passive Elemente sind, die durch Anweisungen be- beziehungsweise verarbeitet werden. In datenflussorientierten Programmiersprachen bestimmen die Daten den Zeitpunkt der Ausführung. De Operationen warten, bis an allen ihren Eingängen Daten angekommen sind, verarbeiten dann die Daten und legen auf jeden Ihrer Ausgänge genau ein Datum. Ausführungsreihenfolge der Operationen wird durch die Verbindungen zwischen den Aus- und Eingängen bestimmt. Ein einfaches Beispiel findet sich in Abbildung 2.7.

Aus diesem Ansatz ergeben sich einige Vorteile. Der Programmieranfänger muss keine Konzepte wie Variablen oder dynamische Datenstrukturen erlernen. Strukturiertes Programmieren wird durch eine mühelose Top-Down-Zerlegung erleichtert. Es kann implizit eine Parallelisierung vorgenommen werden. So können im Beispiel auf Bild 2.7 die beiden cos und das + parallel ausgeführt werden, ohne dass irgendwelche Nebeneffekte zu befürchten sind. Dagegen muss für strukturierte Daten und Ablaufstrukturen wie zum Beispiel Schleifen ein erhöhter Aufwand getrieben werden.

### 2.5.3 Ansätze zur Implementierung

Zur Implementierung von Compilern und Interpretern existieren gut erforschte Datenstrukturen und Enwurfsmuster, die von vielen Compilern genutzt werden. In Appel (2002), Watt und Brown (2000) und Kastens u. a. (2006) werden diese Strukturen und deren Benutzung beschrieben. Diese Konzepte sollen hier kurz zusammengefasst und erläutert werden.

#### Abstrakter Syntaxbaum

In Compilern von verbalen Sprachen hat der abstrakte Syntaxbaum seinen angestammten Platz. Hier wird der Syntaxbaum vom Parser aufgebaut und enthält die Terminale als Blätter. Nichtterminale bilden dabei die Knoten. Danach wird der abstrakte Syntaxbaum mehrmals durchlaufen, um die Knoten mit Daten zu füllen bis zum Schluss am Wurzelknoten das ausführbare Programm zur Verfügung steht. Bei einer visuellen Programmiersprache läuft der Vorgang ein wenig anders ab. Da visuelle Programmiersprachen in der Regel mit einem Editor erstellt werden, wird der Syntaxbaum schrittweise erstellt und editiert, entsprechend der Arbeit mit dem Editor. Während in einem normalen Compiler der Syntaxbaum nur die Terminale explizit enthält, benutzt der Editor den Syntaxbaum zur Berechnung der visuellen Darstellung und legt in dem Baum entsprechende Daten über Position, Größe und Aussehen der Knoten ab.

Bei der Kodegenerierung verhält sich der Syntaxbaum einer visuellen Programmiersprache wieder wie der einer verbalen Programmiersprache. In beiden wird der Baum so oft durchlaufen, bis ausreichend viele Daten zur Kodeerzeugung des Programms zur Verfügung stehen. Der Kode kann Maschinenkode zur nativen Ausführung, Bytecode zur Ausführung auf eine virtuellen Maschine beziehungsweise Interpreter oder auch Quelltext zur weiteren Übersetzung sein.

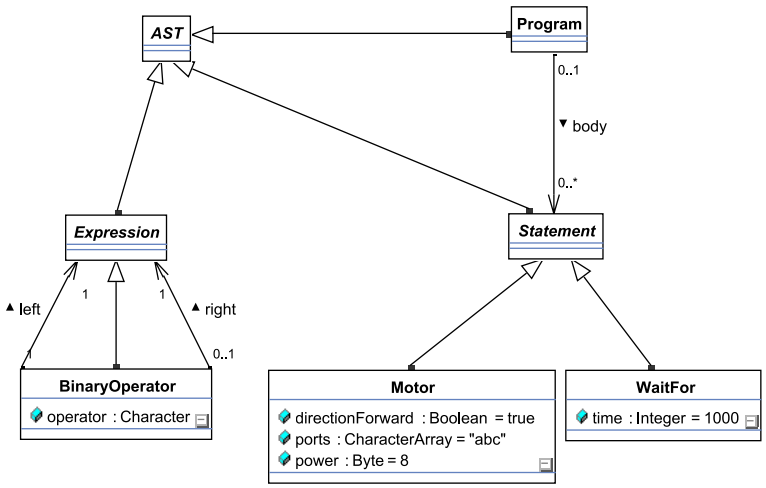


Abbildung 2.8: Aufbau eines abstrakten Syntaxbaumes

In Bild 2.8 auf Seite 19 ist eine Implementierung eines abstrakten Syntaxbaumes zu sehen. Alle Knoten des Baumes erben von der abstrakten Klasse `AST`. In dieser Klassen können Attribute untergebracht werden, die von fast allen Knoten des Baumes verwendet werden, wie zum Baispiel `code` für die Kodefragmente und `x`, `y`, `width`, `height` für die Berechnung der Visualisierung. Weitere Details zur Attributierung von Syntaxbäumen können sind in Kapitel 2.5.3 auf Seite 21 nachzulesen.

Um die Sprachelemente gruppieren zu können, existieren in der Regel noch die abstrakten Klassen `Statement` und `Expression` für Anweisungen und Ausdrücke. Diese abstrakten Klassen dienen der Markierung der Gruppenzugehörigkeit von konkreten Klassen, wie zum Beispiel `Motor`. Die Markerklassen werden auch benutzt, um festzulegen, welche Gruppe von einem Attribut verwendet wird, wie zum Beispiel `BinaryOperator`.

Die Terminale wurden als Attribute in die Klassen integriert, was für eine verbale Programmiersprache unüblich ist. In einer verbalen Programmiersprache wären alle Terminale Texte, die noch in die passenden Datentypen umgewandelt und auf Typkorrektheit geprüft werden müssen. Bei visuellen Programmiersprachen wird diese Aufgabe in der Regel vom Editor übernommen und somit können gleich die Daten im korrekten Typen in die Attribute eingetragen werden.

### Besucher-Muster

Für die Bearbeitung des Syntaxbaumes fehlt jetzt noch das geeignete Werkzeug. Um aus dem abstrakten Syntaxbaum nun Code zu generieren, muss der Baum mehrere Male durchlaufen werden um Informationen zu sammeln oder Bedingungen zu testen. Bei verbalen Programmiersprachen werden klassischerweise Symboltabellen erstellt und dabei wird der Syntaxbaum einige Male durchlaufen. Symboltabellen sind notwendig, um zwischen der Verwendung und der Deklaration eines Bezeichners eine Relation herzustellen und diese auf Korrektheit zu prüfen. Dies bei einer visuellen

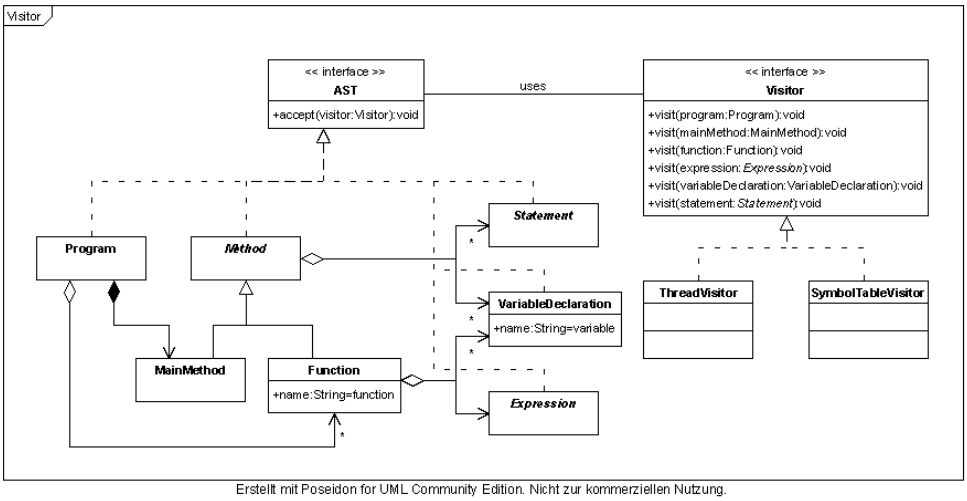


Abbildung 2.9: Das Besucher-Muster

Programmiersprache nicht notwendig, da diese Aufgabe vom Editor übernommen werden kann. Der Editor erstellt dann sofort im Syntaxbaum die entsprechenden Relationen und verhindert die inkorrekte Verwendung.

Bei einer visuellen Programmiersprache muss aber immer noch mindestens ein Durchlauf für die Kodegenerierung erfolgen. Die Visualisierung kann ebenfalls mit einem Durchlauf des Syntaxbaumes erfolgen, wenn die grafische Darstellung bei jeder Änderung einmal gelöscht und danach wieder komplett neu gezeichnet wird.

Eine Möglichkeit der Implementierung wäre, für jede Aufgabe eine Funktion in jedem Baumknoten anzulegen, die diese Aufgabe rekursiv erledigt. Dies hätte den Vorteil, dass die Funktionalität in der dazugehörigen Klasse zu finden ist, was aus objektorientierter Sicht zu begrüßen ist. Allerdings wäre der Code über alle Knotenklassen verteilt und somit schwer zu warten (Appel, 2002, Kapitel 4.3).

Dieses Problem lässt sich durch Anwenden des Besucher-Musters lösen (Grand, 1998, Seite 385ff). Dieses Mustert kapselt die Funktionalität in einem eigenem Objekt, dem Besucher (Visitor). In dem Besucher wird für jeden Knoten eine visit-Funktion implementiert, die die für diesen Baumknoten geeigneten Aktionen ausführt. Der Baumknoten wird der Funktion als Parameter übergeben. Es gibt zwei Möglichkeiten, die visit-Funktionen zu implementieren. Zum einen können die visit-Funktionen überladen werden (siehe Bild 2.9 auf Seite 20), zum anderen kann auch der Name der Baumknoten-Klasse mit in den Namen der visit-Funktion eingearbeitet werden, wie zum Beispiel `visitProgram()` oder `visitStatement()`. Jetzt müssen alle Baumknoten die Funktion `accept()` implementieren. Die Funktion bekommt als Parameter ein Besucher-Objekt übergeben und ruft davon die für den aktuellen Baumknoten bestimmte Funktion auf.

Jetzt verbleibt nur noch die Frage wohin der Code für die Navigation, also wie der Baum durchlaufen wird, zugeteilt wird. Einerseits besteht die Möglichkeit den Navigations-Code in der accept-Funktion unterzubringen (Grand, 1998, Seite 389). Das

hat den Vorteil, dass der Navigations-Code für unterschiedliche Besucher nutzbar ist. Für jede Art des Baumdurchlaufs wird eine eigene accept-Funktion zur Verfügung gestellt. Der Nachteil ist, dass der Source-Code für die Navigation wieder auf alle Baumtypen verteilt wird.

Die andere Möglichkeit ist die Implementierung der Navigation im Besucher (siehe Grand (1998), Seite 391). Dies hat den Nachteil, dass für jeden Besucher der Navigations-Code kopiert wird. Jetzt besteht auch die Möglichkeit, den Durchlauf eines Unterbaumes zu unterbinden, falls in diesem Unterbaum durch den verwendeten Besucher keine Informationen gewonnen werden können.

#### Attributierung des Syntaxbaumes

Die Ergebnisse eines Baumdurchlaufs müssen irgendwo abgelegt werden. Gelegentlich können die Ergebnisse im Besucher als Attribute abgelegt werden, was allerdings selten der Fall ist. In der Regel müssen sie im Syntaxbaum abgelegt werden. Nun könnten für die Ergebnisse entsprechende Attribute in den Baumknoten angelegt werden. Diese Attribute würden jedoch die Baumknoten wieder von Besuchern abhängig machen. Wenn jeder Baumknoten ein assoziatives Feld enthält, könnten die Besucher ihre Ergebnisse als Bezeichner-Wert-Paare ablegen. Wenn jetzt also ein Besucher neue Attribute für seine Ergebnisse benötigt, legt er dynamisch entsprechende Einträge in dem assoziativen Feld an. Somit sind der Syntaxbaum und die Besucher wieder voneinander unabhängig.

## 2.6 DEVIL

DEVIL ist ein Rahmenwerk zur Erstellung von visuellen Sprachen. Es wurde erst nach der Fertigstellung der eigenen Implementierung veröffentlicht und konnte somit nicht verwendet werden. Um die Verwendbarkeit für unsere Sprache zu testen wurde dennoch ein Prototyp erstellt. Der Sprachumfang wurde dazu auf ein Minimum reduziert jedoch musste damit die komplette Werkzeugkette durchlaufen werden. Es musste also ein Programm mit dem generierten Strukturediteor erstellt werden und dann daraus Java-Quelltext generiert werden. Abschließend ist der Java-Quelltext noch mit den LeJOS-Tools zu übersetzen und das Programm auf den Roboter zu übertragen.

### 2.6.1 Wie funktioniert DEVIL?

DEVIL besteht aus einer Sammlung von Sprachen die der Generierung von visuellen Sprachen dienten. Dabei baut das Rahmenwerk auf dem Eli-System auf, welches ein Kompilergenerator für verbale Sprachen ist. An dieser Stelle ist ein deutlicher Schnitt zu erkennen. Während die für DEVIL entwickelten Sprachen in der Regel ohne größere Vorkenntnisse des Kompilerbaus auskommen, sind die von Eli verwendeten Sprachen ohne Kenntnisse in regulären Grammatiken kaum zu verstehen und korrekt anzuwenden.

### Grammatik erstellen

DEVIL verwendet Grammatiken um die Struktur der Sprache zu beschreiben. In der Regel ist die Erstellung einer Grammatik der erste Schritt zu einem visuellen Strukturediteor.

Dazu werden die Klassen des abstrakten Syntaxbaumes definiert. Dies ermöglicht eine intuitivere Darstellung der Grammatik anstelle der regulären Grammatiken. Jede Klasse der Grammatik enthält Attribute zur Speicherung von unterschiedlichen Daten. Es gibt drei unterschiedliche Typen:

**SUB** stellt einen Unterbaum im Syntaxbaum dar. Man kann SUB für jede beliebige selbst definierte Klasse verwenden. SUBs sind gleichzeitig Listen. Durch die Angabe von Zeichen wie *, !, und % lässt sich die Anzahl der Elemente in der Liste kontrollieren. Dabei wird unterschieden, ob diese Konsistenzbedingungen während des Editierens und während der Kompilierung gelten müssen.

**VAL** speichert den Wert eines Basistypen, wobei diese mehr als nur Zahlen und Zeichenketten umfassen. So könnten zum Beispiel die Drehrichtung, der Anschluss und die Geschwindigkeit eines Motors in geeigneten VAL-Attributen abgelegt werden. Diese Werte können unter Verwendung von INIT initialisiert werden und durch die Verwendung von CHECK auf ihre Konsistenz hin überprüft werden.

**REF** ist eine Referenz auf eine Klasse. Ein klassisches Beispiel für eine Referenz ist der Zusammenhang zwischen einer Funktion und deren Aufruf. Die Klasse des Funktionsaufrufs würde eine Referenz auf die aufzurufende Klasse enthalten.

Zusätzlich erlaubt DEVIL noch die Verwendung von mehrfach-Vererbung. Man kann durch INHERITS die Attribute einer Klasse erben. Anders herum kann in einer Liste eine Oberklasse verwendet werden und dadurch können auch alle Unterklassen in diese Liste mit aufgenommen werden. So ist zum Beispiel der Körper eines Programms eine Liste von Anweisungen. Beispiele einer Anweisung sind Motor, WaitForTime und Stop. Somit können alle diese Anweisungen im Körper des Programms verwendet werden.

Die Beschreibung der Grammatik findet in einer Datei mit der Endung model, im Prototypen robolab.model, statt. Mit einer Grammatik erzeugt DEVIL einen Strukturediteor, mit dem ein Syntaxbaum erzeugt, bearbeitet, geladen und gespeichert werden kann. Somit kann die Grammatik vorab schon ein wenig getestet werden.

### Visualisierung erstellen

Der visuelle Teil der visuellen Programmiersprache kann mit Hilfe von visuellen Mustern beschrieben werden. Dazu wird mit Hilfe von visuellen Mustern das Verhalten der einzelnen Klassen und Attribute beschrieben. Dabei ist jedoch zu beachten, dass

nur Klassen über die Toolbar erzeugt werden können und Attribute nur über Dialoge verändert werden können. Um das Verhalten von Robolab nachzuahmen, müssen zum Beispiel die Anschlüsse und Geschwindigkeit eines Motors als eigene Grammatikklassen angelegt werden, die als Unterbäume (SUB) mit dem Motor verbunden werden. Eine Verwendung von Attributen für Anschlüsse und Geschwindigkeit würde es verhindern, die Anschlüsse und Ports über die Toolbar zu erzeugen und dem Motor hinzuzufügen.

Der visuelle Teil der Sprachbeschreibung wird in der Sprache Lido beschrieben und in Dateien mit der Endung lido abgelegt. Diese Sprache ist von Eli übernommen worden und um die visuellen Muster erweitert worden. Somit ist es nicht verwunderlich, dass zwar die visuellen Muster in DEVIL dokumentiert sind, jedoch die Anleitung zu Lido in Eli zu finden ist. Dadurch verwendet die Anleitung einen Sprachschatz der zu DEVIL nicht richtig passt.

Die visuellen Muster werden im Zusammenhang mit so genannten Symbolen verwendet. Ein Symbol für das Attribut time der Klasse WaitForTime lautet programView_WaitForTime_time. Der erste Teil ist der Name der Lido-Datei, der in jedem Symbol als Prefix verwendet wird. Danach kommt der Klassenname gefolgt vom Attributnamen.

Diese Symbole erben von den visuellen Mustern. In der Regel erbt ein Symbol von zwei Mustern. Ein Muster legt fest, wie das Symbol im Editor dargestellt werden soll. Das zweite Muster wird von dem Muster bestimmt, dessen Element dieses Symbol ist. Zum Beispiel erben alle Elemente des Listen-Musters von VPSimpleListElement.

Für die Visualisierung der Grammatik wurden die folgenden visuellen Muster verwendet. Wobei es sich nur um eine Auswahl der zur Verfügung stehenden Muster handelt.

**VPRootElement** ist die visuelle Repräsentation des Wurzelknotens des Syntaxbaumes. Das VPRootElement wird in jeder visuellen Repräsentation mindestens einmal verwendet.

**VPSimpleForm** ist ein Formular. Dies ist das gebräuchlichste Muster in DEVIL, da es mit einer generellen Zeichnung hinterlegt wird. Somit kann hierdurch einer Klasse oder einem Attribut eine Zeichnung zugewiesen werden. Die Hauptaufgabe eines Formulars besteht darin, die Attribute einer Klasse an den in der generellen Zeichnung durch einen Container angegebenen Stelle zu zeichnen. Dazu muss jedes Attribut von VPSimpleFormElement erben und zusätzlich angeben, in welchen Container es gezeichnet werden soll.

**VPSimpleList** ist eine einfache Liste, die horizontal oder vertikal orientiert sein kann. Die Elemente der Liste müssen von VPSimpleListElement erben. Einer Liste können beliebig viele Elemente hinzugefügt, gelöscht und verschoben werden, weshalb sie fast immer im Zusammenhang mit unbeschränkten Teilbäumen (SUB) verwendet werden.

**VPContainer** ist ein Container für ein anderes visuelles Element. Dieser Container ist nicht mit den Containern in den generellen Zeichnungen zu verwechseln.

Ein Container muss im Gegensatz zum Formular nicht mit Werten belegt sein, er kann auch leer sein. Dann erscheint an Stelle des VPContainerElement eine hellgelb gefärbte Fläche. Die Farbe dieser Fläche ist für alle Container konfigurierbar.

**VPImagePrimitive** ist ein Bild. Es ist der Pfad zu einer Bilddatei anzugeben.

**Generelle Zeichnungen** Generelle Zeichnungen werden immer im Zusammenhang mit dem Formular-Muster verwendet. Hierdurch werden sie zu einem sehr mächtigen und häufig eingesetzten Muster.

Eine generelle Zeichnung ist eine Vektorzeichnung, lässt sich also ohne Qualitätsverluste skalieren. Gleichzeitig werden in den Zeichnungen so genannte Container angelegt, die als Platzhalter für andere visuelle Muster fungieren. Da die genaue Größe der Platzhalter zum Zeitpunkt der Erstellung der Zeichnung nicht feststeht, lassen sich horizontale und vertikale Dehnungsintervalle festlegen. Das Verhalten der Dehnungsintervalle läßt sich noch durch eine Vielzahl weiterer Parameter einstellen.

Neben den Containern können in den generellen Zeichnungen die bekannten grafischen Primtive wie Linien, Pfeile, Kreise, Ovale, Rechtecke und Texte verwendet werden. Diese Primitive lassen sich durch etliche Parameter den eigenen Wünschen anpassen. Leider ist die Auswahl an Farben auf schwarz und weiß begrenzt. Im Dialog lassen sich noch nicht mal die RGB-Werte direkt angeben.

**Sichten** Sichten visualisieren Teilbäume des mit der Grammatik möglichen Syntaxbaumes. So ist es zum Beispiel möglich in einer Sicht eine Übersicht der vorhandenen Funktionen und Variablen anzubieten und für jede Funktion eine eigene Sicht zu öffnen. Hierdurch wird das Programm in handliche Einheiten geteilt und unterstützt visuell die Trennung der lokalen Variablen in den Funktionen voneinander.

Sichten werden in der model-Datei angelegt. Es wird für jede Sicht der Name festgelegt, der gleichzeitig der Name der für diese Sicht zu benutzenden Lido-Datei ist und somit auch zum Prefix der Symbole in dieser Datei wird. Weiterhin werden die Toolbar-Knöpfe für die Sicht hier definiert.

Die Toolbar-Knöpfe gibt es in unterschiedlichen Ausführungen. In der einfachen Ausführung bestehen sie nur aus einem Text für den Knopf sowie der zu erzeugenden Klasse. Wenn statt des Textes eine Bilddatei angegeben wird, erhält der Knopf das Bild als Piktogramm. Gleichzeitig kann auch ein Text für eine Ballonhilfe angegeben werden.

Wenn der Platz für die Knöpfe in der Werkzeugleiste nicht mehr ausreicht können auch mehrere Spalten angelegt werden. Meistens geht bei vielen Knöpfen die Übersicht verloren und die Knöpfe können mit Hilfe von Menüknöpfen gruppiert werden. So könnten die Knöpfe für die Geschwindigkeit von Motoren hinter einem Menüknopf für die Geschwindigkeit zusammengefasst werden.

### Initialisierung, Konsistenzüberprüfung und Synchronisation erstellen

Bei jeder Sprache müssen Syntax und Semantik korrekt sein, damit der Quelltext übersetzbar ist. Dazu überprüft der Kompiler in mehreren Stufen den Quelltext auf Korrektheit und gibt gegebenenfalls entsprechende Fehlermeldungen heraus. Zwar unterstützen moderne Editoren und Entwicklungsumgebungen den Entwickler durch Syntaxhighlightning und Vervollständigung von Schlüsselwörtern, Funktionsnamen und deren Parametern bei der korrekten Erstellung von Quelltext, jedoch muss der Kompiler weiterhin den Quelltext komplett auf Korrektheit überprüfen.

DEVIL ist dagegen anders aufgebaut. Der Editor baut automatisch im Hintegrund einen Syntaxbaum auf und kann während des Editierens Syntax und Semantik überprüfen und im Fehlerfall umgehend einen Fehler melden. Somit weiß der Entwickler sofort ob er etwas falsch gemacht hat und muss nicht erst den Quelltext abspeichern und übersetzen.

DEVIL bietet dazu die Möglichkeit der Initialisierung, der Snychronisation und Konsistenzprüfung an, die in den folgenden Unterabschnitten erklärt werden.

**Initialisierung** Die Initialisierung dient der Bestückung von Sprachelementen mit sinnvollen Standardwerten. Dies hilft auf jeden Fall schnell ein lauffähiges Programm zu erzeugen, da vom Entwickler diese Werte nicht von Hand gesetzt werden müssen. Allerdings müssen die Standardwerte in der Regel bearbeitet werden, damit sich ein funktionsfähiges Programm ergibt.

**Synchronisation** Während die Initialisierung nach dem Erzeugen eines Sprachelements durch Standardwerte für einen konsistenten Zustand sorgt, hält die Synchronisation während des Editierens die Synax- und Semantikregeln ein. Somit lässt sich zum Beispiel garantieren, dass ein Motor nie mehr als drei Ports verwendet und jeder Port auch nur einmal verwendet wird. Dabei werden in der Regel keine Fehlermeldungen ausgegeben sondern es werden ohne Mitwirken des Benutzers direkt im Syntaxbaum Modifikationen vorgenommen.

**Konsistenzprüfungen** Konsistenzprüfungen werden vor der Quelltextgenerierung und nach der Bestätigung von Dialogen ausgeführt. Diese Prüfungen enden im Fehlerfall in einer entsprechenden Meldung und erfordern das Eingreifen des Benutzers.

### Erzeugung des Quelltextes

In der Regel generieren die mit DEVIL erzeugten Editoren keine ausführbaren Programme sondern Quelltext. Dies bietet den Vorteil dass der Editor unabhängig von Änderungen in der Codeerzeugung wird und zudem die Optimierung nicht selbst geleistet werden muss. Bei der Verwendung von Quelltext kann unter Umständen auch eine Unabhängigkeit von der verwendeten Plattform erreicht werden.

Zur komfortablen Erzeugung von Quelltext bietet DEVIL die IPTG-Bibliothek an. Der Entwickler definiert Quelltextschnipsel, in denen durch Platzhalter weiterer Text

eingefügt wird. Aus diesen Quelltextschnipseln werden PTG-Strukturen generiert, mit denen sich schnell ein Baum zusammensetzen lässt. Dies geschieht schnell und effizient, da die PTG-Strukturen nur miteinander verbunden werden, jedoch die eigentliche Generierung des Quelltextes erst bei der Ausgabe in die Datei.

### 2.6.2 Tcl/Tk

DEVIL kann mit verschiedenen Sprachen erweitert werden, am besten sind C/C++ und Tcl/Tk eingebunden. Tcl ist zur Erweiterung von DEVIL besonders gut geeignet, da die Visualisierung und der Editor Tk verwenden. Tcl wird interpretiert und schlägt deshalb bei der Erzeugung eines Editors nicht zu Buche. Das ist beim Debuggen hilfreich, da so eine schnelle Turn-Around-Zeit erreicht wird. Tcl verfügt über eine wirksame Grabage-Collection, was den Umgang mit dynamischen Variablen vereinfacht. Diesen Vorteilen steht gegenüber, dass Tcl interpretiert wird und deshalb nicht so schnell ist wie C/C++. Da die mit der visuellen Programmiersprache erstellten Programme einen geringen Umfang haben werden, ist die Geschwindigkeit nicht entscheidend. Deshalb wurde für sämtliche Erweiterungen von DEVIL Tcl verwendet.

#### Tcl Grundlagen

Tcl ist als ein Ersatz für die Skript-Sprachen der Shells wie bash oder csh geschrieben worden. Daher weist Tcl einige Eigenschaften auf, die bei Sprachen wie C nicht zu finden sind. Tcl ist von Haus aus eine imperative Sprache, die eine hohe Dynamik aufweist. Das heißt, es kann zur Laufzeit vom Programm aus der Code eingesehen und modifiziert werden. Unter Tcl ist es eine gängige Praxis, die Existienz von Variablen und Prozeduren zu überprüfen, Prozeduren zu überschreiben oder eine Befehlszeile in einer Variable zu erzeugen und diese dann ausführen zu lassen. Durch diese Möglichkeiten kann Tcl nach Belieben angepasst werden und eine nachträgliche Erweiterung um eine Objektorientierung stellt kein Problem dar.

Auf Grund der Verwandschaft mit den Skriptsprachen der Shells kennt Tcl keine Datentypen. Alle Variablen sind Strings. Zusätzlich bietet Tcl noch Listen und assoziative Felder an. Auf den ersten Blick wirkt dieses übersichtliche Angebot an Datentypen und Datenstrukturen einschränkend, jedoch lässt sich hiermit und mit ein wenig Kreativität nahezu jede beliebige Struktur nachbilden. Dabei erweist sich das assoziative Feld als Geheimwaffe, was in Kapitel 4.5 zu sehen ist.

Tcl gibt es für nahezu jedes Betriebssystem, angefangen von Windows und Linux bis hin zu Handies und PDAs. Tcl abstrahiert dabei die konkrete Plattform und bietet standardisierte Schnittstellen zu den gebräuchlichsten Betriebssystemfunktionen an. Dadurch ist es mit nur geringem Aufwand möglich, ein Tcl-Programm zu entwickeln, das auf nahezu jedem Betriebssystem läuft.

Für geschwingkeitskritische Probleme und zur Einbindung als Skriptsprache in Anwendungen bietet Tcl ein umfangreiches Interface zu C/C++ und anderen Sprachen an. Auf diese ist Tcl auch als Skriptsprache für DEVIL und ELI eingebunden

worden.

### Oberflächenprogrammierung

Die Bibliothek zur Erzeugung von grafischen Benutzeroberflächen heißt Tk. Tk gibt es wie Tcl für fast jedes Betriebssystem und wird deshalb von vielen anderen Sprachen, wie zum Beispiel Python, ebenfalls benutzt. Tk bietet die gebräuchslichten GUI-Elemente an, fortgeschrittenere Komponenten wie Bäume oder Tabellen fehlen. Diese lassen sich durch eine große Auswahl an Bibliotheken nachrüsten. Die grafischen Komponenten heißen unter Tk Widgets.

Die Manipulationsmöglichkeiten für Widgets sind sehr umfangreich und ermöglichen dem Programmierer eine weitgehende Anpassung des Aussehens der Oberfläche an seine Wünsche. Etwas gewöhnungsbedürftig ist die Tatsache, dass alle Routinen zur Ereignisbehandlung im globalen Kontext ausgeführt werden und deshalb kein Zugriff auf lokale Variablen möglich ist. Bei der Übergabe von Parametern kann dieses Problem noch übergangen werden, jedoch muss zur Übergabe von Rückgabewerten eine globale Variable bemüht werden, wenn der Rückgabewert nicht in ein Widget eingetragen werden kann.

# 3 Entwurf

## 3.1 Vorarbeiten

Zu Beginn der Arbeit wurde ein grober Entwurf erstellt, wie die visuelle Programmiersprache später aussehen soll und mit welchen Bibliotheken die Sprache implementiert werden soll.

Zuerst wurde festgelegt, welchen Rahmen die visuelle Programmiersprache erfüllen soll. Es standen dazu die beiden visuellen Programmiersprachen von LEGO zur Verfügung: RCX code und ROBOLAB. Beide Sprachen folgen dem Konzept des Steuerflusses. Das heißt, die Ausführung der Anweisungen wird durch den Programmzähler kontrolliert. Dieses Konzept ist für eine visuelle Programmiersprache nicht optimal, eine datenflussorientierte Sprache wäre viel besser zu handhaben.

Die Vorteile einer datenflussorientierten visuellen Programmiersprache sind, dass keine Variablen benötigt werden. Die Daten fließen entlang den Verbindungen von einem Block zum anderen. Dort werden die Daten verarbeitet und dann zum nächsten Block weitergeleitet.

Trotzdem folgt die visuelle Programmiersprache dem Konzept des Steuerflusses, da die Sprache im didaktischen Gesamtkonzept zu sehen ist. Ziel des Informatikunterrichtes ist es unter anderem, den Schülern eine verbale, objektorientierte Sprache beizubringen und ihnen die damit verbundenen Problemstellungen der Informatik nahe zu bringen. Am häufigsten wird Java als Programmiersprache verwendet. Diese weist durch ihre Abstammung von C noch eine starke Steuerflussorientierung auf.

Die zu entwickelnde viuselle Programmiersprache soll nun aber nicht ein weiteres Sprachkonzept einführen sondern einen einfachen Einstieg in die im Informatikunterricht gebräuchlichen Konzepte bieten. Dies ist nur mit einer steuerflussorientierten visuellen Programmiersprache möglich.

Die Objektorientierung der visuellen Programmiersprache konnte nicht realisiert werden. Alle bekannten objektorientierten visuellen Sprachen haben zwei Probleme:

1. Um die Relationen zwischen den Klassen und Objekten darzustellen sind sehr viele Verbindungen notwendig. Schon kleine Programme enthalten so viele Verbindungen, dass ein intuitives Verstehen des Diagramms nicht mehr gewährleistet ist.

2. Es werden immer zwei Sichten auf das Programm benötigt, die statische und die dynamische Sicht. In der statischen Sicht werden die Klassen und deren Relationen untereinander beschrieben, in der dynamischen Sicht das Verhalten

der Klassen. Dieses Konzept ist von Programmieranfängern schwer zu verstehen.

Somit wurde entschieden, dass die visuelle Programmiersprache rein imperativ sein sollte und keine Objektorientierung enthält. Mit dieser Sprache sollen nur einfache Programmierkonzepte wie Schleifen, Prozeduren und Variablen unterrichtet werden.

Robolab bietet mit den Level ein gutes Konzept, um die Schüler schrittweise an den Sprach- und Funktionsumfang von Robolab heranzuführen. Jedoch haben die Level von Robolab den Nachteil, dass sie nicht konfigurierbar sind. Somit müssen sich die Unterrichtskonzepte an den Vorgaben von Robolab orientieren oder den Schülern wird ein größerer Sprachumfang zugemutet als eigentlich notwendig, da ein Sprachkonstrukt fehlt, was nur in einem höheren Level zur Verfügung steht. Um diese Probleme zu lösen, sollten die Level zu Profilen erweitert werden, die eine freie Konfigurierbarkeit des Sprach- und Funktionsumfanges ermöglichen.

Eine Implementierung des Editors für die visuelle Programmiersprache war ursprünglich in Java vorgesehen. Bei der Suche nach Bibliotheken zur Unterstützung der Implementierung ist der Autor auf DEVIL gestoßen. DEVIL hat den Vorteil, dass es die komplette Infrastruktur wie Syntaxbäume, Visualisierung des Syntaxbaumes, Codegenerierung und Konsistenzbedingungen bereits zur Verfügung stellt, während sie unter Java noch mühevoll implementiert und getestet hätte werden müssen. Es war jedoch nicht klar, ob DEVIL die Anforderungen bezüglich Anpassbarkeit erfüllen würde. Deshalb wurde ein Prototyp entwickelt.

## 3.2 Prototypen

Um die Verwendbarkeit von DEVIL im Zuge dieser Diplomarbeit zu prüfen, sollte ein Prototyp erstellt werden, der das Verhalten von Robolab nachbildet. Zusätzlich sollte die Unterstützung von LeJOS getestet werden und es musste die Machbarkeit des neuen Profil-Konzeptes überprüft werden.

### 3.2.1 1. Prototyp

Für den ersten Prototyp wurde ein reduzierter Sprachumfang gewählt, da die Implementierung der Sprache nicht der zu untersuchende Punkt war. Vielmehr wird davon ausgegangen, dass sich die Sprache mit Devil realisieren lässt und deshalb keiner weiteren Untersuchung bedarf. Interessanter ist vielmehr die Frage, ob es möglich ist die externen Tools zur Kompilierung und Übertragung des Programms auf den Roboter in den Editor zu integrieren. Die Profile mussten außerdem in adäquater Weise umgesetzt werden.

Folgende Sprachelemente werden umgesetzt:

- Motor vorwärts mit den Geschwindigkeitsstufen eins bis fünf und den Ports A bis C.

- Stop mit den Ports A bis C.
- Warten mit der zu wartenden Zeit in Millisekunden.

### Grammatik

Um ein ähnliches Verhalten wie in Robolab zu erzielen wurde eine formlose Grammatik basierend auf dem Mengen-Muster (`VPSet`) erstellt. Hier ist eine grafische Darstellung zu sehen:

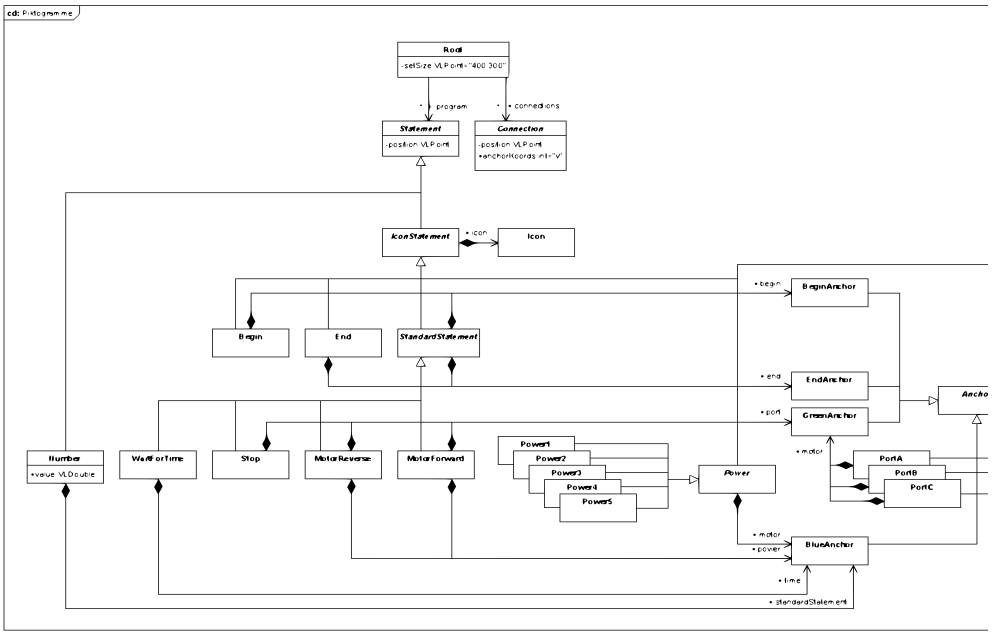


Abbildung 3.1: Formlose Grammatik für den ersten Prototyp, Teil 1

Die Root-Klasse ist der Wurzelknoten des Programms und verwaltet sämtliche Klassen. Die Wurzelklasse enthält das von der Visualisierung geforderte Attribut `setSize`. Die Funktion dieses Attributs wird in Kapitel 3.2.1 näher beschrieben.

Das Attribut `setSize` wird mit dem Tupel (400, 300) initialisiert. In der Grammatik geschieht dies mit der `INIT`-Klausel. Des weiteren ist dieses Attribut als privat markiert. Dadurch soll verdeutlicht werden, dass dieses Attribut nicht durch einen Dialog editierbar ist. Es wurde diese Darstellung gewählt da das verwendete UML-Tool keine äquivalente Darstellung anbietet. Alle Attribute die per Dialog editierbar sind wurden als öffentlich markiert.

Die Wurzelklasse enthält noch die zwei Listen `program` und `connections`. Die letzte Liste enthält sämtliche Verbindungen zwischen den Programmen. Die erste Liste enthält das Programm und besteht aus `Statements`. Die Visualisierung fordert von der Klasse `Statement` das Attribut `position`. Die Klasse Statement dient als Oberklasse für sämtliche frei positionierbaren Sprachobjekte, also die Piktogramme und Zahleneingaben. Sie ist genauso wie in der Grammatik als abstrakt markiert.

Von der Klasse `Statement` erbt die abstrakte Klasse `IconStatement`. Diese Klasse repräsentiert alle Piktogramme der visuellen Sprache. Die Visualisierung erforderte, dass das Piktogramm in die Klasse `Icon` ausgelagert wird. Die Klassen `Begin` `End` und die abstrakte lasse `StandardStatement` sowie die `Power`- und `Port`-Klassen erben von `IconStatement`.

Die Klassen `Begin` und `End` sind die Piktogramme für den Anfang und das Ende eines Programms wie sie auch in Robolab verwendet werden. Sie enthalten jeweils einen Anker der Endpunkt einer Verbindung ist. Die Klasse `Begin` besitzt mit dem Attribut `begin` einen Anker der Klasse `BeginAnchor` und die Klasse `End` besitzt mit dem Attribut `end` einen Anker der Klassen `EndAnchor`. Eine Verbindung erfolgt immer vom `BeginAnchor` zum `EndAnchor`. Nähere Einzelheiten zur Verwendung zweier unterschiedlicher Anker für eine Verbindung werden bei der Beschreibung der Verbindungen weiter unten erläutert.

Die abstrakte Klasse Standardstatement besitzt über die Attribute `begin` und `end` jeweils einen `BeginAnchor` und einen `EndAnchor`. Diese Standardsprachelemente befinden sich demnach also zwischen zwei Verbindungen, da eine Verbindung ankommt und eine andere abgeht. Von ihr erben die Klassen `MotorRevers`, `MotorForward`, `Stop` und `WaitForTime`.

Die Klasse `Stop` entspricht dem Sprachelement Stop und besitzt als Attribut den zu verwendenden `port`. Da der Port auch als ein Piktogramm dargestellt werden soll, was mit dem Piktorgramm *Stop* verbunden wird, ist der `port` nicht direkt ein Port sondern ein `GreenAnchor`. Dieser Anker markiert Verbindungen zu Ports.

Die Klassen `MotorForward` und `MotorReverse` repräsentieren die entsprechenden Sprachelemente zur Motorsteuerung. Sie besitzen genau wie die Klasse `Stop` ein `Port`-Attribut und zusätzlich noch ein `power`-Attribut mit einem `BlueAnchor` zur Verbindung mit numerischen Elementen. Auch wenn ein Motor-Element mit einem Nummern-Element verbunden werden kann, werden die Power-Elemente der Einfachheit und Übersichtlichkeit halber bevorzugt verwendet.

Die Klassen `PortA`, `PortB` und `PortC` stehen für die einzelnen Ausgabekanäle und besitzen im Attribut `motor` wie die Klassen `Stop`, `Motorforward` und `MotorReverse` einen `GreenAnchor`. Hierüber kann eine Verbindung zu den eben aufgezählten Klassen der Motorsteuerung hergestellt werden und somit kann diesen Klassen ein Port zugewiesen werden.

Die bevorzugte Methode den Klassen der Motorsteuerung die Geschwindigkeit zuzuweisen ist die Verwendung der Power-Piktogramme. Dazu wurden die Klassen `Power1` bis `Power5` erzeugt, die von der abstrakten Klasse `Power` erben. Dadurch war es in der Visualisierung möglich, sämtliche Eigenschaften bis auf das Bild für das Piktogramm an der Klasse Power zu definieren. `Power` besitzt auch über das Attribut `motor` den Anker `BlueAnchor`. Hierüber werden die Motor-Klassen mit den Power-Klassen verbunden um die Geschwindigkeit festzulegen.

Es verbleibt noch das Standardstatement `WaitForTime`. Es benötigt als Eigenschaft die zu wartende Zeit. Dazu verfügt dieses Sprach-Element durch das Attribut `time` einen `BlueAnchor`. Hierüber wird es bevorzugt mit dem Statement `Number` verbunden, welches mit dem Attribut `standardStatement` ebenfalls über

einen `BlueAnchor` verfügt. Die Klasse `Number` legt in dem Attribut `value` den numerischen Wert ab.

Die Klassen `BeginAnchor`, `EndAnchor`, `GreenAnchor` und `BlueAnchor` werden von der abstrakten Klasse `Anchor` beerbt . In `Anchor` werden sämtliche visuelle Eigenschaften bis auf die Farben der Anker definiert.

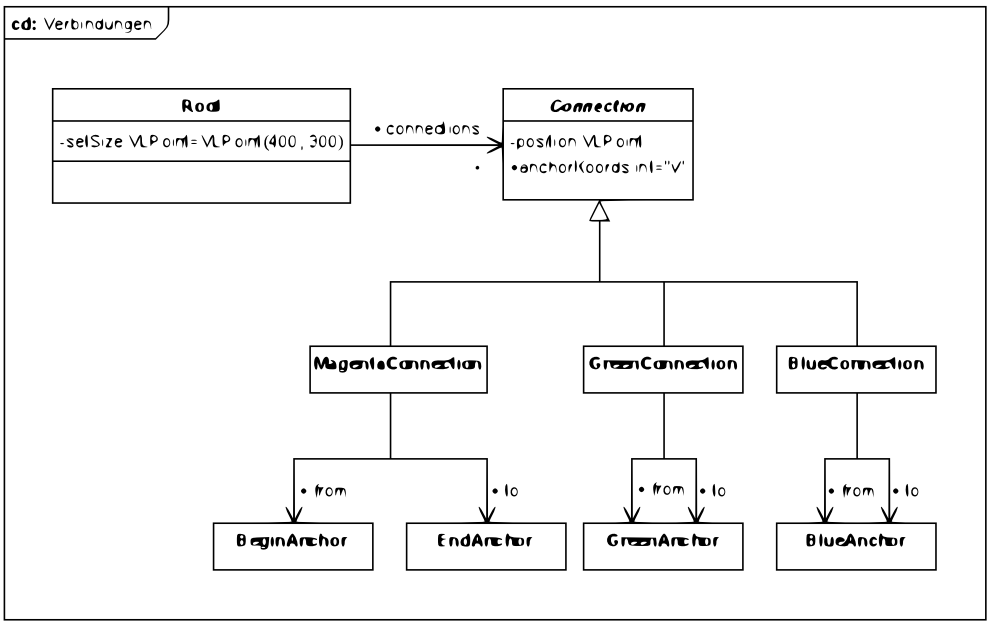


Abbildung 3.2: Formlose Grammatik für den ersten Prototyp, Teil 2

Die abstrakte Klasse `Connection` erfüllt die Voraussetzungen der visuellen Muster mit ihren Attributen `position` und `anchorKoords`. Die Klassen `MagentaConnection`, `GreenConnection` und `BlueConnection` erben von ihr. Der Kontrollfluss wird durch die `MagentaConnection` dargestellt, die `Greenconnection` stellt Verbindungen zwischen den Ports und den Klassen für die Motorsteuerung her. Die `BlueConnection` dient der Zuweisung von numerischen Werten.

Die Verbindungen besitzen jeweils die Attribute `from` und `to`, die Referenzen auf entsprechende Anker repräsentieren. Die blaue und grüne Verbindung verfügt nur über eine Anker-Klasse, so dass die Attribute `from` und `to` auf die gleiche Anker-Klasse verweisen. Hierdurch ist nicht kontrollierbar, in welche Richtung der Pfeil der Verbindung zeigt, was bei diesen Verbindungen im Prinzip auch unwichtig ist. Beim Kontrollfluss ist die Richtung der Pfeile jedoch von Bedeutung und deshalb haben dort die Attribute `from` und `to` jeweils ihren eigenen Anker erhalten. Es bleibt noch die Selbstverständlichkeit anzumerken, dass eine Verbindung nur zwischen den entsprechenden Ankern hergestellt werden kann. So können keine grünen Verbindungen zwischen blauen Ankern hergestellt werden.

### Visualisierung

Wie bei Devil üblich wird die Visualisierung in einem separaten Schritt erledigt. Wie in Kapitel 2.6.1 auf Seite 22 beschrieben werden Symbole verwendet. Jedes Symbol erbt dabei in der Regel von zwei Mustern. Die Beschreibungen der Muster in diesem Kapitel wurden sinngemäß der Dokumentation von Devil (siehe DEVIL Benutzerhandbuch (2007)) entnommen.

Das Symbol `programView_Root` erbt das visuelle Muster `VPRootElement`. Dieses Wurzelelement-Muster positioniert das Objekt in der linken oberen Ecke der Darstellungsfläche. Jedes Wurzel-Symbol erbt von diesem Muster. Das zweite Muster, von dem dieses Symbol erbt, ist `VPSingletonForm`. Dieses Muster ist ein Spezialfall des Formularmusters. Es zeichnet ein Rechteck mit genau einem Element, in diesem Fall das Symbol `programView_Root_program`. Ausnahmsweise erbt dieses Symbol noch von einem dritten Muster, `VPConnectionArea`. Dieses Muster bestimmt den Bereich in dem Verbindungen gezeichnet werden dürfen. Die Verbindungen werden in dem Attribut `connections` abgelegt.

Das Symbol `programView_Root_program` steht für das Attribut `program` der Klasse `Root`. Es erbt von `VPSingletFormElement` und macht dadurch deutlich, dass es das einzige Element von `VPSingletonForm` ist. Das zweite Muster ist `VPSet`. Dieses Mengenmuster ermöglicht die beliebige Positionierung der Mengenelemente innerhalb des Zeichenbereichs. Alle Mengenelemente werden mit `VPSetElement` markiert. Das Besondere an diesem Muster ist, dass die Größe und die Position des Darstellungsbereichs in den Attributen `setSize` und `position` gespeichert werden. Da in diesem Prototypen die Zeichenfläche das ganze Fenster ausfüllen soll, ist die Position nicht von Interesse und wurde deshalb nicht als Attribut in der Klasse `Root` angelegt. Das Attribut `setSize` wurde in der Klasse `Root` angelegt und mit der Mindestgröße initialisiert.

Das Symbol `programView_Root_connections` speichert eine Liste der Verbindungen und erbt deshalb von `VPConnectionList`. Dies ist eine Anforderung des Verbindungs-Musters `VPConnectionArea`. Da die Darstellung von `VPConnectionArea` übernommen wird, braucht dieses Symbol keine weiteren Muster.

Die Anker erben in der Grammatik alle von der Klasse `Anchor`. Deshalb werden alle visuellen Eigenschaften bis auf die Farbe im Symbol `programView_Anchor` festgelegt. Da ein Anker ein Endpunkt einer Verbindung ist, erbt das Symbol von `VPConnectionEndpoint`, was Bestandteil des Verbindungs-Musters ist. Um einen Anker darstellen zu können, wird ein entsprechendes Muster benötigt. Zu diesem Zweck erbt das Symbol `programView_Anchor` von `VPVectorPrimitive`, das Muster für grafische Primitive. Hiermit wird der rechteckige Anker gezeichnet. Dazu werden die temporären Attribute `SYNTH.size` für die Größe und `SYNTH.outlineWidth` zum Abschalten des Rahmens gesetzt. Da der Anker das Element des Container-Musters aus den Klassen `Begin`, `End` oder `StandardStatement` ist, erbt er noch von `VPContainerElement`.

Die Spezialisierungen der Klasse `Anchor` definieren in ihren Symbolen nur noch die entsprechenden Farben. Die restlichen Eigenschaften erhalten sie durch Vererbung

in der Grammatik. Für programView_BeginAnchor und programView_EndAnchor wird die Farbe auf Magenta gesetzt, für programView_GreenAnchor auf Grün und für programView_BlueAnchor auf Blau.

Alle Statements sind Mengenelemente des Mengen-Musters, und deshalb erbt das Symbol programView_Statement von VPSetElement. Sie selbst folgen dem Formular-Muster und erben VPForm. Die zu verwendende generelle Zeichnung wird noch nicht festgelegt und ist von den Spezialisierungen dieser Klasse zu definieren.

Durch die Spezialisierung IconStatement wird nur das Icon zum Statement hinzugefügt. Deshalb muss auch nur das Symbol programView_IconStatement_icon weiter definiert werden. Dieses Symbol erbt von VPFormElement, da es ein Element des Formularmusters von Statement ist. Es wird festgelegt, dass dieses Formularelement in den Container icon der generischen Zeichnung zu zeichnen ist. Zur Darstellung erbt programView_IconStatement_icon von VPImagePrimitive. Dieses Muster stellt ein Bild dar. Das Bild ist noch in einer Spezialisierung anzugeben.

Das Symbol programView_StandardStatement erbt nicht direkt von einem Muster sondern indirekt über Grammatik-Vererbung. Es legt aber die generelle Zeichnung für das Formular-Muster fest.

programView_StandardStatement_begin und programView_StandardStatement_end sind Elemente des Formular-Musters und erben entsprechend. Sie sind in der generellen Zeichnung in den Containern begin und end einzusetzen. Da diese Symbole die Anker enthalten, erben sie von dem Muster VPContainer. Das Container-Muster bietet Platz für maximal ein Element und stellt dieses gegebenenfalls dar. Sollte kein Element vorhanden sein, wird eine rechteckige Fläche dargestellt.

Die Klassen Begin und End erben von der Klasse IconStatement. Deshalb werden zunächst nur die Symbole der Klasse Begin betrachtet, da End ähnlich aufgebaut ist. Auf Grund der Grammatik-Vererbung wird die generelle Zeichnung (Symbol programView_End) und das zu verwendende Bild für das Piktogramm (Symbol programView_End_icon) spezifiziert. Analog zum StandardStatement werden die Anker festgelegt, allerdings hat jede Klasse nur einen der beiden Kontrollflussanker.

Die Klassen zur Motorsteuerung werden analog zu den beiden im letzten Absatz beschriebenen Klassen mit einer Visualisierung versehen. Es werden wieder die generelle Zeichnung und das Bild für das Piktogramm festgelegt. Die Anker für den Kontrollfluss müssen nicht weiter spezifiziert werden, da dies schon in der Eltern-Klasse StandardStatement geschehen ist. Jedoch werden Anker für die Verbindungen für die Port-Piktogramme und numerischen Ausdrücke angelegt. Dies geschieht jedoch analog zu den Kontrollfluss-Ankern in StandardStatement.

Bei den Port- und Power-Klassen werden zur Visualisierung ähnliche Techniken angewendet wie bei den anderen Klassen die von IconStatement abgeleitet sind. Auch bei den anderen Klassen die von StandardStatement erben werden die gleichen Mechanismen angewendet wie bei den Klassen zur Motorsteuerung.

Dies gilt auch für die Klasse Number, obwohl sie nur von der Klasse Statement erbt. Die visuelle Beschreibung folgt dem gleichen Schema wie bei IconStatement, da der Unterschied nur das Piktogramm ist, was diese Klasse nicht benötigt.

Abschließend werden noch die Verbindungen erklärt. Durch das Symbol program

`View_Connections` wird für die Eltern-Klasse `Connections` festgelegt, dass sie dem rechtwinkligen Verbindungsmuster folgt, in dem das Symbol von `VPOrthoConnection` erbt. Gleichzeitig wird die Linienstärke der Verbinder auf 2 festgesetzt. Über die Vererbung der Grammatik werden diese Eigenschaften für alle Verbindungsklassen festgelegt.

Am Beispiel der Verbindungsklasse für den Kontrollfluss soll jetzt erläutert werden, wie die Visualisierung spezifiziert wird. Dies gilt analog für die anderen Verbindungsklassen. Mit dem Symbol `programView_MagentaConnection` wird die Farbe der Verbindung festgelegt. Da jede Verbindung eine andere Farbe hat, konnte dies nicht in der Eltern-Klasse festgelegt werden. Die Symbole `programView_Magenta Connection_from` sowie `programView_ MagentaConnection_to` erben von `VPConnectionFrom` und `VPConnectionTo`, und legen damit den Anfang und das Ende der Verbindung fest.

Wie zu sehen ist, kann durch geschickte Anwendung der Vererbung in der Grammatik viel Arbeit in der Visualisierung erspart werden. Vor allem können damit Konfigurationspunkte zentralisiert werden.

### Profile

Die Profile sind eine Weiterentwicklung des Level-Mechanismus von Robolab. Im Gegensatz zu Robolabs statischen Levels lassen sich die Profile vom Benutzer konfigurieren. Es können sowohl neue Profile erzeugt werden als auch der Umfang für jedes Profil festgelegt werden.

Das Hauptproblem bei Devil war, einen geeigneten Einstiegspunkt für die Anzeige der Profil-Dialoge zu finden. Zum einen ist es sehr einfach möglich, eigene Menüpunkte in das Untermenü von Process zu integrieren indem ein weiterer Prozessor hinzugefügt wird der eine Tcl/Tk-Funktion aufruft. Diese Möglichkeit bietet sich für die Konfiguration der Profile an, jedoch wurde im Prototyp davon kein Gebrauch gemacht. Hierdurch erfolgt auch keine Lösung des Problems, dass beim Starten des Editors das Profil ausgewählt werden muss.

Um dieses Problem zu lösen kann aus dem globalen Kontext von Tcl/Tk eine Funktion aufgerufen werden, die die entsprechende Oberfläche zum Abfragen des zu verwendenden Profils erzeugt. Der Nachteil ist jedoch, dass Devil noch so gut wie gar nicht initialisiert ist und solche Dienste wie die Fehler-Protokollierung noch nicht initialisiert sind.

Deshalb bietet Devil seit der Version 0.15.0 die Möglichkeit, dem Initialisierungsprozess weitere Funktionen hinzuzufügen. Dies geschieht indem mit

```
main::addInitProc INIT_FUNCTION
```

eine weitere Funktion dem Initialisierungsprozess hinzugefügt wird. Zum Zeitpunkt des Aufrufs der neuen Initialisierungsfunktion sind zumindest die für die Abfrage des zu benutzenden Profils wichtigen Dienste wie `CDlg` zur einfachen Erstellung von Dialogen und `log` zur Fehlerprotokollierung initialisiert.

Um die Implementierung des Prototypen schnell voran zu treiben, wurde die Konfiguration der Profile in einem Unterdialog von der Abfrage des zu benutzenden Profils beim Starten des Editors verlegt, anstelle die Konfiguration in einem eigenen Menüpunkt auszugliedern. Im Falle einer kompletten Implementierung mit Devil sollte es keine Probleme bereiten, die Konfiguration in einen eigenen Untermenüpunkt zu verlegen.

Jetzt fehlt nur noch die Möglichkeit, in der Toolbar die Einträge für die Sprachelemente zu verstecken oder anzuzeigen, je nachdem wie es in dem aktiven Profil konfiguriert worden ist. Leider existiert dafür in Devil noch kein passendes Interface, weshalb man sich mit einer wenig eleganten Methode behelfen muss. In Devil gibt es die Funktion `meToolbar::create`, die der zentrale Einsprungspunkt für die Erzeugung der Knöpfe auf der Toolbar ist. Diese Funktion wird mit einer modifizierten Variante überschrieben, die für jeden Knopf in der Profilkonfiguration nachschaut, ob dieser angezeigt werden soll oder nicht.

Diese Methode ist umständlich, da der Quelltext von der Originalfunktion `meToolbar::create` kopiert und anschließend modifiziert wird. Dieser Schritt muss für jede neue Devil-Version wiederholt werden, da sonst eventuelle Änderungen in der Originalfunktion verloren gehen.

Somit sind alle Probleme seitens Devil gelöst, jedoch kennt Tcl/Tk keine Strukturen wie C, sondern nur Listen und assoziative Felder. Somit mussten die Konfigurationsdaten in einem assoziativen Feld abgelegt werden, in dem der Index des Feldes aus dem Profilnamen und dem Namen des Sprachelementes mit Hilfe eines Punktes als Trennzeichen zusammengesetzt wird.

Durch Lösen dieser Probleme ist der Nachweis erbracht worden, dass sich die Profile unter Devil implementieren lassen. Auf die Implementierung einer hierarchischen Darstellung der Sprachelemente, der Speicherung der Konfiguration auf der Festplatte sowie die Verschlüsselung des Passwortes für die Konfiguration wurde verzichtet, da diese Punkte keine prinzipiellen Probleme mehr aufwerfen.

### Kodegenerierung

Die Kodegenerierung unter Devil funktioniert im großen und ganzen ähnlich wie bei der Kodegenerierung in Robolab (siehe Kapitel 2.5.3). Auch Devil durchläuft den Syntaxbaum und baut aus einzelnen PTG-Strukturen den Quelltext auf. Deshalb wird auf die Funktionsweise von PTG und die Erstellung von Quelltext nicht weiter eingegangen, da sich der Unterschied lediglich auf die Verwendung der Sprache LIDO anstelle von Java beschränkt.

Devil ist darauf angewiesen, dass zur Kodegenerierung ein ordentlicher Syntaxbaum existiert. Das heißt, dass die einzelnen Knoten durch `SUB`-Schlüsselwörter miteinander verknüpft sind. Dies ist jedoch bei der in diesem Prototypen verwendeten Grammatik nicht der Fall. Es existiert nur eine unsortierte Liste von Anweisungen, deren Reihenfolge vom zeitlichen Ablauf beim Einfügen und Löschen abhängig ist und in keinem Zusammenhang mit dem Kontrollfluss steht. Deshalb wird die korrekte Reihenfolge der Anweisungen nicht eingehalten und der generierte Quelltext

ist nicht verwendbar.

In der Grammatik wird der Kontrollfluss durch die magenta-farbenen Verbindungen sichergestellt. Leider ist Devil nicht in der Lage, diese Verbindungen korrekt auszuwerten. Auch ist Devil nicht in der Lage das Start-Element ausfindig zu machen.

Die entsprechende Logik könnte nun allerdings hinzugefügt werden. Die erste Methode wäre, die komplette Kodegenerierung unter Umgehung von Devil selber zu implementieren. Der Aufwand dafür ist hoch und es stellt sich die Frage, was für ein Vorteil dann gegenüber einer Implementierung in Java bleibt. Hier befindet man sich in einer Sackgasse.

Die zweite Möglichkeit wäre, einen für Devil geeigneten Syntaxbaum aufzubauen und aus diesem mit Devils Bordmitteln Quelltext zu erstellen. Diese Methode hätte gegenüber der ersten den Vorteil, dass auf diesen Syntaxbaum wieder die Mittel der Konsistenzprüfung und Synchronisierung anwendbar wären, wenn die Erstellung des Syntaxbaumes in Echtzeit möglich wäre. Dies stellt auf Grund der zu erwartenden Größe der Programme für den RCX kein Problem dar.

Dazu müsste jedoch nach jeder Editieroperation der Syntaxbaum verworfen und neu erzeugt werden. Dafür wären die entsprechenden Funktionen von Devil zu überschreiben. Durch so eine starke Überarbeitung von Devil würde sich die Versionabhängigkeit jedoch so stark erhöhen, dass die überschriebenen Funktionen wahrscheinlich für jede Devil-Version gepflegt werden müssten.

Zusammenfassend lässt sich sagen, dass dieser formlose Grammatikansatz zwar visuell nahe an Robolab ist, jedoch kaum Vorteile gegenüber einer Java-Implementierung bringt. Devil ist für diese Art Grammatiken nicht gedacht. Es sollte deshalb ein zweiter Prototyp gebaut werden, der eine formale Grammatik verwendet.

### 3.2.2 2. Prototyp

Leider erwies sich die formlose Grammatik als Sackgasse. Zwar wird durch die formlose Grammatik die höchste Ähnlichkeit mit Robolab erreicht, jedoch kann Devil hieraus nur unter großem Aufwand Quelltext generieren. Deshalb soll für den zweiten Prototypen eine formale Grammatik erstellt werden. Für den Struktureditor bedeutet das, dass die Piktogramme nicht mehr frei positionierbar sind. Die Position der Piktogramme wird von Devil berechnet und lässt sich vom Benutzer des Struktureditors nicht beeinflussen. Auch das Verbinden der Piktogramme entfällt, da die Verbindungen von der Grammatik nicht mehr benötigt werden.

Um zumindest die optische Ähnlichkeit mit Robolab zu erhalten, sollen die Verbindungen durch gezeichnete Linien simuliert werden. Somit wird sich ein Prototyp ergeben, der sich wie RCX-Code benimmt, jedoch optisch aussieht wie Robolab. Dies hat den Vorteil, dass sich keine Fehler mehr durch fehlende oder falsche Verbindungen ergeben können.

### Grammatik

Die Grammatik des zweiten Prototypen unterscheidet sich nur in einem Punkt wesentlich von der Grammatik des ersten Prototypen. Beim zweiten Prototypen fehlen die Verbindungen und damit auch die Anker. Dies sorgt dafür, dass die Grammatik des zweiten Prototypen wesentlich übersichtlicher ist.

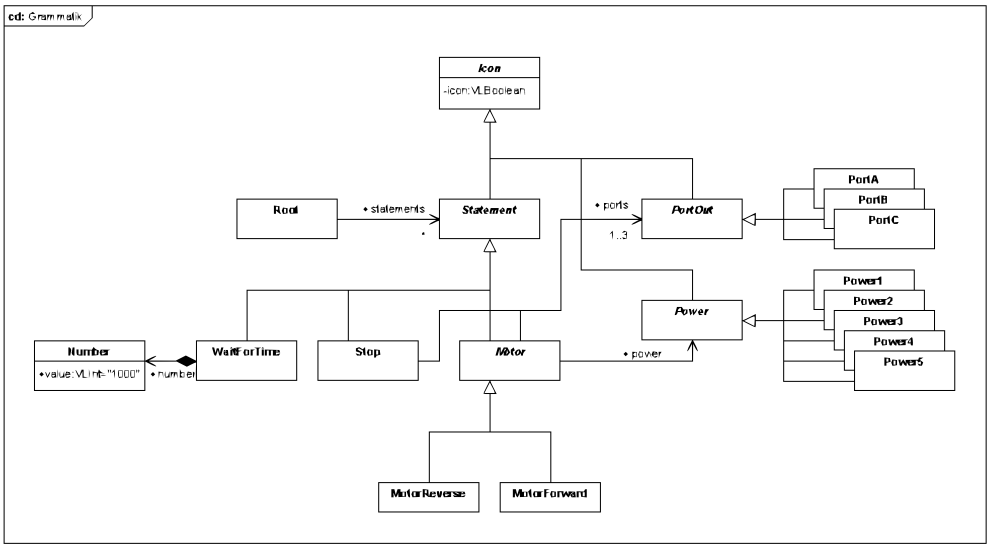


Abbildung 3.3: Die Grammatik des zweiten Prototypen

Auch in dieser Grammatik gibt es die Wurzelklasse `Root`. Diese verwaltet in der Liste `statements` sämtliche Anweisungen des Programms. Der Kontrollfluss des Programms folgt der Reihenfolge der Anweisungen in dieser Liste, somit sind für den Kontrollfluss keine Verbindungen notwendig. Auch die Klassen `Begin` und `End` konnten weggelassen werden, da sie immer den ersten und letzten Platz in der Anweisungsliste belegen würden. Stattdessen werden die Piktogramme immer mit der grafischen Repräsentation der Wurzelklasse gezeichnet.

Die abstrakte Klasse `Statement` repräsentiert sämtliche Anweisungen und erbt von der abstrakten Klasse `Icon`, die das Bild für das Piktogramm hält. Das Attribut `icon` der Klasse `Icon` ist nur definiert worden um ein Symbol für die Darstellung des Piktogramms zu haben. Alle Piktogramme wie `PortOut` und `Power` erben von `Icon`.

Die Klasse `Motor` ist die abstrakte Oberklasse zu `MotorReverse` und `MotorForward` und enthält sämtliche benötigten Attribute wie `ports` und `power`. Durch diese konsequente Vererbungsstrategie ist eine Zentralisierung der visuellen Beschreibung möglich, da diese sich ebenfalls entlang der Grammatikvererbung fortpflanzt. Der gleiche Mechanismus wurde auch für die von `Power` abgeleiteten Geschwindigkeitsklassen und die von `PortOut` abgeleiteten Portklassen angewendet.

Die Klassen `Stop`, `WaitForTime` und `Number` sind analog zu denen im ersten Prototypen aufgebaut bis auf die Tatsache, dass die Anker für die Verbindungen fehlen.

### Visualisierung

Die Visualisierung hat sich gegenüber dem ersten Prototypen durch den Einsatz des Listen- anstelle des Mengen-Musters wesentlich geändert.

Das Symbol `programView_root` der Wurzelklasse setzt das Formularmuster ein. Die verwendete generelle Zeichnung beinhaltet die Start- und Ende-Piktorgramme. Das Attribut `statements` wird durch das Listen-Muster `VPSimpleList` visualisiert.

Auch das `Icon` ist ein Fomular, in das durch das Attribut `icon` ein Bild gezeichnet wird. Durch die Vererbung in der Grammatik brauchen alle Klassen die von `Icon` erben keine Muster mehr für das Piktogramm zu spezifizieren. Sie müssen nur noch das für das Piktogramm zu verwendende Bild und die zu verwendende generelle Zeichnung festlegen.

Das Symbol der Klasse `Statement` erbt von `VPSimpleListElement`. Es ist die Aufgabe der Klasse ein Element der Anweisungsliste zu sein.

Die Klasse `Motor` erbt sämtliche visuellen Muster durch die Grammatik-Vererbung. Das Symbol `programView_Motor` legt nur noch die zu verwendende Zeichnung fest, die für alle Motoren gleich ist. Das Attribut `ports` ist ein Formularelement von `Motor` und eine Liste von Ports. Die Menge der Ports auf mindestens einen und höchsten drei einzuschränken kann hier nicht geleistet werden und muss in einer Tcl-Funktion realisiert werden (siehe Kapitel 3.2.2). Da das Attribut `power` nur ein Piktogramm enthalten kann, erbt dieses vom Container-Muster. Das Singleton-Muster lässt sich hier nicht verwenden, da das Singleton-Muster immer ein Element enthalten muss, während das Container-Muster auch leer sein darf.

`MotorForward` und `MotorReverse` erben alle benötigten Muster über die Grammatik und müssen somit nur noch die Bilder der Piktogramme an den Symbolen `programView_MotorForward_icon` und `programView_MotorReverse_icon` definieren.

Die Klasse `Stop` wird analog zu der `Motor`-Klasse definiert bis auf die Tatsache, dass kein Attribut für die Geschwindigkeit vorhanden ist. `WaitForTime` nutzt die gleiche generelle Zeichnung wie `Stop` und das Attribut `number` ist analog zu `power` aus `Motor`.

Die Klasse `PortOut` ist ein Listen-Element des Attributs `ports` der Klassen `Motor` und `Stop`. Ansonten werden `PortOut` und die von ihr erbenden Klassen behandelt wie die anderen Piktogramme auch.

Die von `Power` abgeleiteten Geschwindigkeitsklassen sind Container-Elemente, ansonsten gelten hier auch die Regeln für Piktogramme.

Die Klasse `Number` ist ein Sonderfall, da sie noch nicht mal ein Piktogramm ist. Daher erbt sie vom Containerelement und vom Formular-Muster. In das Muster wird dann der Wert des Attributs `value` unter der Verwendung des Text-Primitivs `VPValTextPrimitive` geschrieben.

### Profile

Die Profile wurden vom ersten Prototypen übernommen und nicht weiterentwickelt, da schon im ersten Prototypen die Integrierbarkeit der Profile gezeigt wurde.

### Kodegenerierung

Die Kodegenerierung ließ sich mit der formalen Grammatik ohne größere Probleme implementieren. Einzig die Klassen zur Motorsteuerung benötigten erhöhte Aufmerksamkeit.

Die Motorklassen besitzen eine Liste von Ports, worauf der Steuerbefehl angewendet wird. Das heißt, mit einer Anweisung lassen sich mehrere Motoren steuern. LeJOS kann das nicht. Im Gegenteil, die Bibliothek benötigt mehrere Anweisungen um Motor A vorwärts drehen zu lassen:

```
Motor.A.setPower(5);
Motor.A.forward();
```

Diese Anweisungsliste muss für jeden Port wiederholt werden. Somit müssen diese Anweisungen in der Portliste erzeugt werden. Es wird für diesen Quelltextschnipsel Zugriff auf die Geschwindigkeit benötigt. Diese ist jedoch von dem Symbol `Motor_ports` aus nicht verfügbar. Vom Symbol `Motor` aus wären zwar beide Attribute zugreifbar, jedoch werden die Ports dann als gesamte Liste zurückgegeben und sind nicht mehr einzeln auflösbar.

Der erste Lösungsversuch bestand in der Verwendung einer Tcl-Funktion im Symbol `Motor`. Die Funktion `gencode::motor` erzeugt den kompletten Quelltext zur Steuerung sämtlicher Ports, jedoch geht dabei die korrekte Einrückung des Quelltextes verloren.

Nach Rücksprache mit Devil-Entwicklern ergab sich ein besserer Lösungsweg. So wurde das Symbol `Motor_power` als Attribut in das Symbol `Motor` geholt. Nun wurde im Symbol `Motor_ports` für jeden Port oben genannter Quelltextschnipsel erzeugt. Die Geschwindigkeit ist jetzt im Symbol `Motor` mit der LIDO-Anweisung `INCLUDE` abrufbar. Die PTG-Strukturen werden im Symbol `Motor` mit einem `CONSTITUENTS` miteinander verknüpft.

**Konsistenzbedingungen** Eine Motoranweisung darf zum Zeitpunkt der Quelltextgenerierung höchstens auf drei und mindestens auf einem Port wirken, auch darf jeder Port nur einmal vorkommen. Während des Editierens darf die Portliste ruhig leer sein, da dadurch die Handhabung vereinfacht wird.

Alle Anweisungen zur Motorsteuerung werden defaultmäßig mit allen drei Ports initialisiert. Dafür ist die Funktion `edithook::create_Motor` zuständig. Somit ist die Portliste nach der Initialisierung konsistent.

Während des Editierens ist die Funktion `synchook::Motor.ports` für die Einhaltung der Konsistenzbedingungen zuständig. Diese Funktion überprüft, dass höchstens drei unterschiedliche Ports in der PortListe sind, es dürfen keine Ports doppelt

in der Liste sein. Dazu wird die Liste sortiert. Wenn dabei zwei gleiche Ports gefunden werden, wird einer verworfen. Nach dieser Operation können nicht mehr als drei Ports in der Liste sein, da es nur drei unterschiedliche Ports gibt und doppelte Einträge entfernt worden sind.

Bevor die Kodegenerierung aufgerufen wird überprüft die Funktion `checkutil::NumberOfPorts`, ob mindestens ein Port angegeben worden ist.

Für die Motoren muss auch immer eine Geschwindigkeit angegeben sein. Während des Editierens kann die Geschwindigkeit auch fehlen.

Die konsistente Initialisierung wird von der Funktion `edithook::create_Motor` übernommen, indem sie die Geschwindigkeit auf Maximum setzt. Vor der Kodegenerierung überprüft Devil auf Grund der in der Grammatik angegebenen Kardinalitäten die Konsistenz.

#### Compilierung und Übertragung zum Roboter

Die Übertragung zum Robotor geschieht durch einen Eintrag im Menü *Process*. Dieser Eintrag wird durch die Zeile

```
file::registerProcessor robot genCode::transferToRCX
  "Transfer to RCX"
```

erzeugt und an die Funktion `genCode::transerToRCX` gebunden. Diese Funktion ruft erst die Kodegenerierung durch die Funktion `c::startProces- sor genCode $root` auf. Danach werden mit der Tcl-Anweisung `exec` die Programme `lejosjc`, `lejoslink` und `lejosdl` der Reihe nach aufgerufen. Die jeweils nächste Anweisung wird nur dann aufgerufen wenn die vorherige erfolgreich ausgeführt wurde.

### 3.2.3 Ergebnisse

Abschließend lässt sich sagen, dass sich die Aufgabenstellung dieser Arbeit auch mit Devil implementieren lässt. Die Befürchtung, dass sich das Profil-Konzept nicht verwirklichen lässt, erwies sich als unbegründet. Allerdings musste dafür eine Implementierung in Kauf genommen werden, die bei jeder neuen Devil-Version Pflege benötigt. Möglicherweise gibt es demnächst für diese Art von Erweiterungen ein Interface.

Für den Prototypen sind die Profile nur konzeptuell implementiert worden. Es fehlt noch eine hierarchische Darstellung der Sprachelemente. Dafür ist eine Baumkomponente notwendig, die von Tk nicht zur Verfügung gestellt wird. Die Baumkomponente muss von einer Bibliothek zur Verfügung gestellt werden. Damit ergibt sich ein Installationsproblem, da die Erweiterung in den Installationsprozess des Editors eingefügt werden muss. Dies sollte sich jedoch lösen lassen.

Alle weiteren fehlenden Eigenschaften der Profile sollten sich ohne Probleme implementieren lassen und eher eine Fleißarbeit denn eine Herausforderung darstellen.

Auch die Werkzeugkette vom visuellen Programm bis zum Übertragen von Java-Bytecode auf dem RCX ließ sich ohne größere Probleme realisieren. Um die Verwendung der Werkzeugkette einfacher zu gestalten, fehlen noch geeignete Konfigurationsmöglichkeiten um sie an unterschiedliche LeJOS-Installationen anpassen zu können.

Bei der Sprache musste eine Einschränkung hingenommen werden, da Devil nicht für informale Grammatiken gedacht ist. Dadurch fällt die freie Positionierbarkeit der Piktogramme weg. Jedoch war es möglich, das Verhalten von RCX-Code zu erreichen und dabei das visuelle Aussehen von Robolab zu wahren. Allerdings wurde das geänderte Verhalten nicht als Nachteil empfunden sondern macht es den Programmieranfängern einfacher, da es nicht mehr möglich ist, Verbindungen zwischen den Piktogrammen zu vergessen oder die Piktogramme falsch zu verbinden.

Ansonsten hält Devil das Versprechen, die Last des Programmierers von der Implementierung der Infrastruktur auf die Sprachspezifikation zu verlagern. Auch die Trennung von Grammatik, Visualisierung und Kodegenerierung ermöglicht eine Implementierung, die spätere Änderungen auf einen Teil der Implementierung beschränken. Gerade die gut gelungene Abschottung der Kodegenerierung von dem Rest der Implementierung ermöglicht es auf eine andere Zielsprache wie zum Beispiel NQC umzustellen.

Allerdings haben sich auch einige Schwächen von Devil bemerkbar gemacht. Es fehlen in Devil geeignete Schnittstellen um die Editoren mit Funktionalität zu erweitern, insbesondere wenn nicht die Sprache selber erweitert werden soll, wie zum Beispiel bei den Profilen. Auch ein Weg zum Einbringen von Tcl-Bibliotheken ist noch nicht spezifiziert, auch wenn dies durch Handarbeit bei der Erzeugung des Installers möglich ist.

Dabei ist jedoch zu beachten, dass Devil einen sehr stabilen Eindruck hinterlässt. Der generierte Editor hat nur einmal einen Fehler aufgewiesen, der ihn kommentarlos abstürzen ließ. Auch die Entwicklungsumgebung funktioniert reibungslos, wenn gewisse Regeln eingehalten werden.

## 3.3 Visuelle Programmierprache

Nachdem im vorherigen Kapitel geklärt wurde, in welche Entwicklungsumgebung der Editor entwickelt werden soll, wird es jetzt Zeit für den Entwurf der visuellen Programmiersprache. Dazu wird die Sprache Robolabs untersucht und an Hand dessen überlegt, welche Eigenschaften von Robolab übernommen werden sollen und welche Eigenschaften einer Überarbeitung bedürfen.

### 3.3.1 Verbindungen

Verbindungen sind in den meisten visuellen Programmiersprachen ein Problem. Häufig werden zur Erstellung eines nicht trivialen Programms viele Verbinder benötigt,

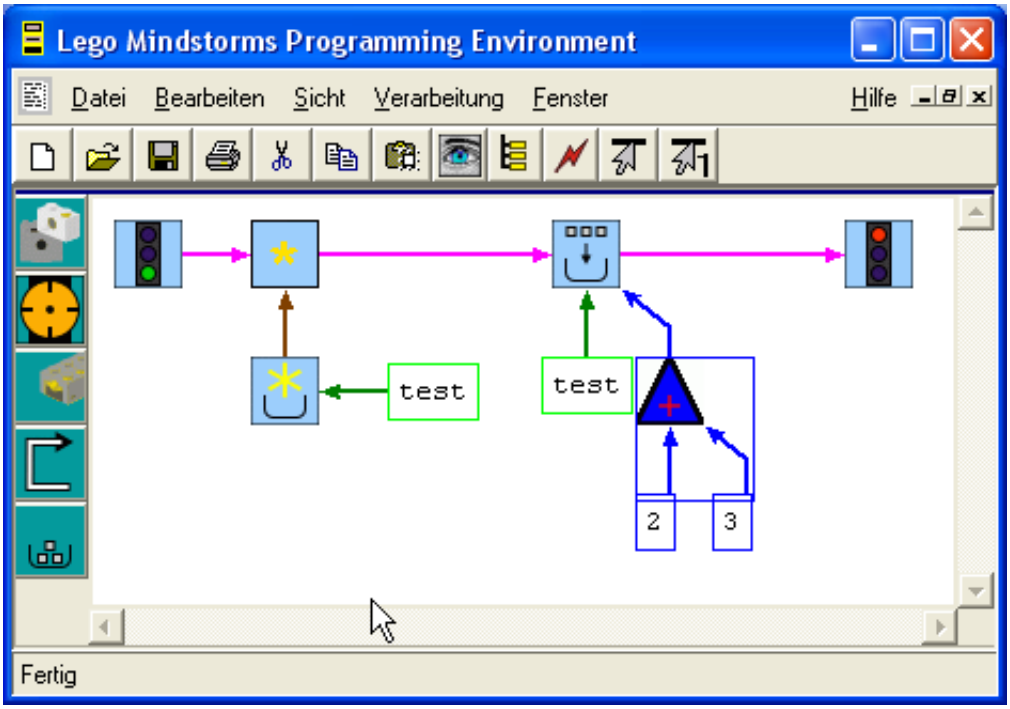


Abbildung 3.4: Die von der Programmiersprache verwendeten Verbindungen

so dass sich diese oft kreuzen oder andere Sprachelemente gar überqueren. Ein solches Programm ist dann nicht mehr übersichtlich.

Robolab hingegen setzt Verbinder so ein, dass sich ohne Probleme auch größere Programme ohne gekreuzte Verbindungen oder überquerte Piktogramme erstellen lassen. Zum einen wird dies durch das steuerflussorientierte Konzept erreicht, wodurch in der Regel ein Piktogramm nur mit seinem Vorgänger und mit seinem Nachfolger verbunden wird. Weitere Parameter werden unter dem Piktogramm abgelegt. Durch dieses Verfahren fühlt sich der Programmierer nicht veranlasst, sich kreuzende Verbindungen zu erstellen oder eine Verbindung über ein anderes Piktogramm zu legen.

Robolab vermeidet es auch, Verbinder für die Referenzierung von Variablen zu benutzen. Statt dessen werden die Variablen über Bezeichner identifiziert: für die ersten vier Variablen existieren unterschiedlich farbige Piktogramme, die restlichen werden über Zahlen angesprochen. Hierdurch bleiben Verbindungen lokal zu Piktogrammen in der direkten Nachbarschaft. Diese Eigenschaft von Robolab soll beibehalten werden und wird in den Kapiteln 3.3.5 und 3.3.6 entsprechend beachtet.

Die einzelnen Piktogramme eines Diagramms werden durch Pfeile unterschiedlicher Farbe verbunden. Der Steuerfluss wird duch pinke Pfeile dargestellt und verbindet Anweisungen. Der Datenfluss verwendet blaue Pfeile, um Ausdrücke zu verbinden. Die Ein- und Ausgänge und Referenzen werden durch grüne Pfeile verbunden. Braune Pfeile werden für die Elemente der Deklarationsliste verwendet.

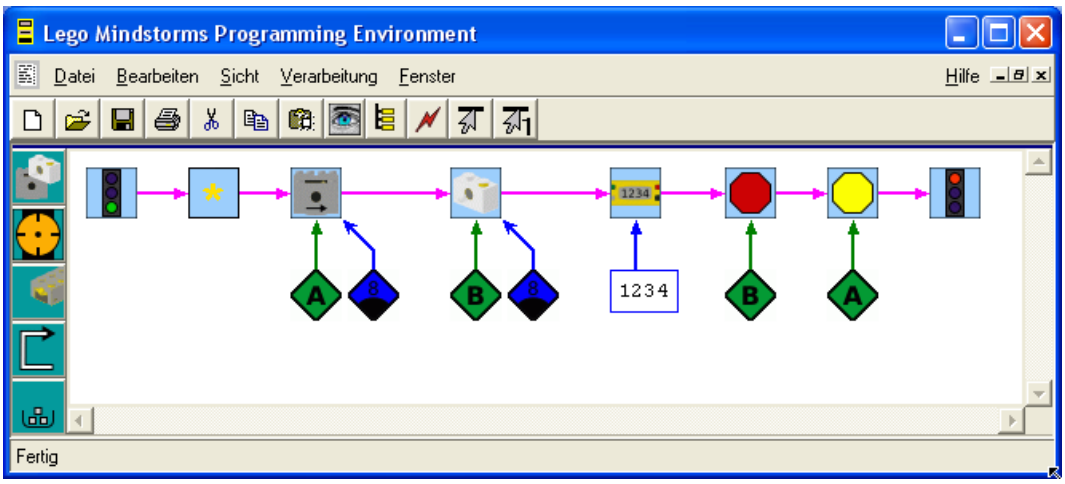


Abbildung 3.5: Hier sind sämtliche Möglichkeiten zur Ausgabe abgebildet. Die Lampe auf Ausgang B wird am Ende ausgeschaltet, der Motor auf Ausgang A läuft aus.

### 3.3.2 Ausgabemöglichkeiten

Hier werden sämtliche Möglichkeiten der Sprache beschrieben, Ausgaben zu tätigen. Neben den Motoren und Lampen umfasst dies auch die LCD-Anzeige des RCX.

Der Motor hat die Parameter Port, Geschwindigkeit und Richtung. Der Port wird durch einen Buchstaben gekennzeichnet. Die Geschwindigkeit ist eine Zahl, und die Richtung wird grafisch durch Pfeile repräsentiert. Im Gegensatz zu Robolab unterstützt LeJOS acht Geschwindigkeiten für den Motor. Diese acht Geschwindigkeitsstufen sollen auch dem Programmierer zugänglich sein.

Die Lampen benötigen die gleichen Parameter wie der Motor, bis auf die Drehrichtung. Somit benutzen die Lampen auch die gleichen Piktogramme wie die Motoren.

LeJOS bietet Klassen, die die Ausgabe auf dem LCD ermöglichen. Es ist sogar unter Einschränkungen möglich, Text auf dem LCD auszugeben. Da unsere Programmiersprache nur ganze Zahlen unterstützt (siehe Kapitel 3.3.5 auf Seite 46), wird auf die Ausgabe von Texten verzichtet und sich auf die Ausgabe von ganzen Zahlen beschränkt. Als Parameter wird ein Ausdruck benötigt.

### 3.3.3 Sensoren

Die *WaitFors* warten auf das Eintreten eines bestimmten Ereignisses. Es sollen *WaitFors* für das Drücken und Loslassen eines Berührungssensors implementiert werden. Bei den Lichsensoren soll das Überschreiten oder Unterschreiten eines Schwellenwertes sowie die Änderung nach oben oder unten überwacht werden. Alle *WaitFors* benötigen einen Eingang als Parameter, bei den Lichtsensoren ist zusätzlich noch ein Schwellenwert nötig. Darüber hinaus gibt es noch Ausdrücke, die die direkte Abfrage eines Sensorwertes zulassen.

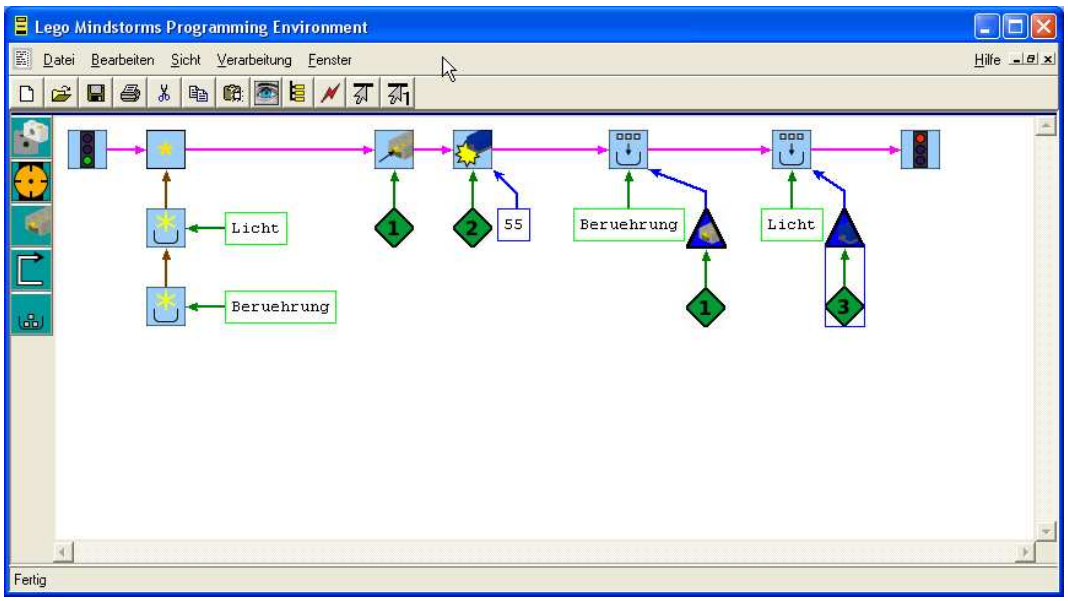


Abbildung 3.6: Warten auf eine Berührung, Warten auf das Überschreiten eines Schwellwertes beim Lichtsensor. Abfragen des Berührungszustandes und des Lichtes

Auf dem RCX sind die Knöpfe *View*, *Prgm* und *Run* vorhanden, die von LeJOS abgefragt werden können. Dies soll ebenfalls unter der visuellen Programmiersprache möglich sein. Dazu gibt es Ausdrücke zur Abfrage des Zustands der Knöpfe.

### 3.3.4 Zeit

Bei der Programmierung des RCX wird viel mit Zeit gearbeitet. Um eine bestimmte Strecke zurückzulegen, werden die Motoren angeschaltet, dann wird eine Zeitspanne gewartet und danach werden die Motoren abgeschaltet. Darüber hinaus gibt es auch noch *Timer*, die Zeiten über mehrere Anweisungen hinweg messen können. Es ist möglich, bis zu einem bestimmten Zeitpunkt zu warten oder direkt den Wert eines *Timers* abzufragen. Selbstverständlich sind auch Anweisungen zum Deklarieren und Zurücksetzen des Timers vorhanden.

### 3.3.5 Variablen und Ausdrücke

Die Programmiersprache soll auch mit globalen Variablen umgehen können. Eine Variable kann ganze, vorzeichenbehaftete Zahlen von 32 Bit Länge verarbeiten. LeJOS kann auch mit Fließkommazahlen von 32 Bit Länge verarbeiten, jedoch werden Programmieranfänger diese nicht benötigen und müssen sich nicht mit den Untiefen von Variablen unterschiedlichen Typs und der Konvertierung dazwischen belasten.

Eine Variable muss deklariert werden. Sie kann gesetzt und abgefragt werden. Zum setzen kann jeder beliebige Ausdruck verwendet werden. Ebenfalls kann das

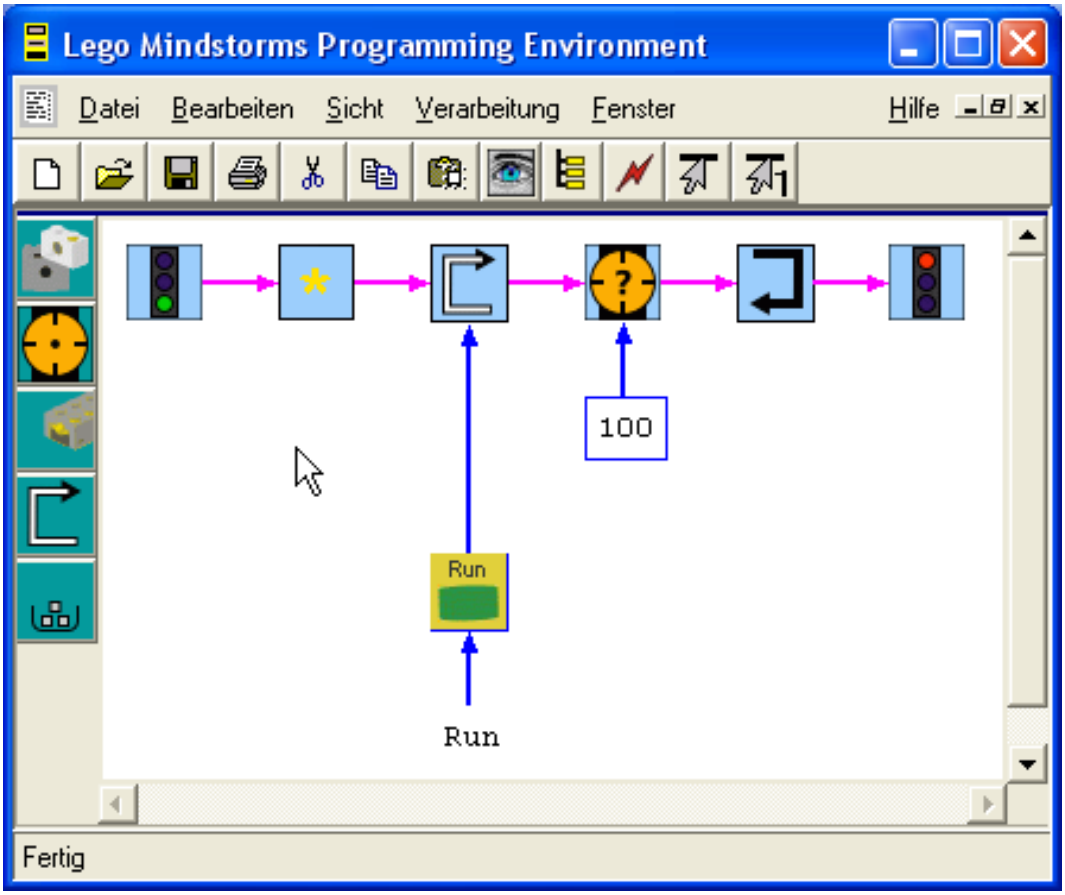


Abbildung 3.7: Warten auf Drücken des *Run*-Knopfes auf dem RCX.

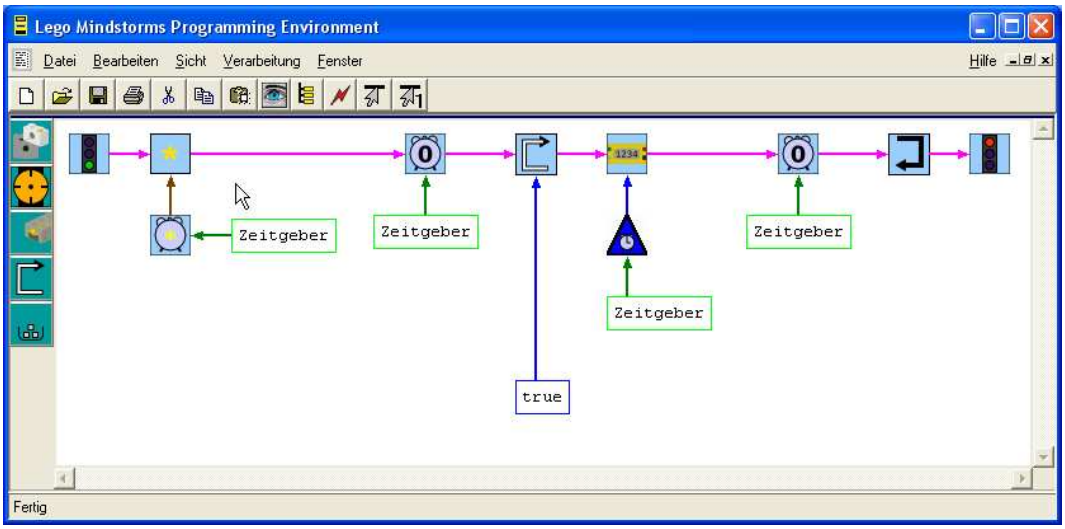


Abbildung 3.8: Programm-Schnipsel zur Abfrage des Timers.

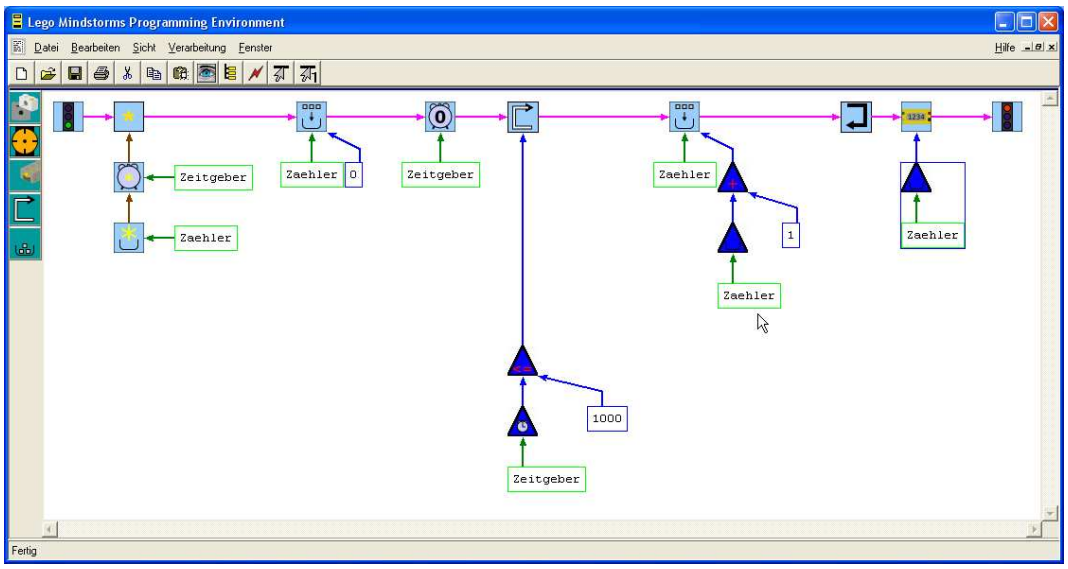


Abbildung 3.9: Variablen deklarieren, setzen und abfragen

Ergebnis der Variablenabfrage in einem Ausdruck verwendet werden. Für Ausdrücke stehen Operationen mit den gängigen Grundrechenarten zur Verfügung.

Für die Kontrollstrukturen stehen auch boolsche Operationen zur Verfügung. Obwohl versucht wurde, eine Sprache mit nur einem Datentypen zu entwickeln, musste dieses Konzept an dieser Stelle aufgegeben werden. Bei nur einem ganzzahligen Datentypen war es nicht möglich, einen für Anfänger verständlichen Java-Quelltext zu erzeugen. Gerade die boolschen Operatoren ließen sich nur schwer auf ganzzahlige Datentypen abbilden.

Somit haben boolsche Operatoren und Vergleiche einen anderen Ergebnistyp als die Operatoren der Grundrechenarten. Vergleiche bilden aus zwei Zahlen einen boolschen Wert und boolsche Operatoren verknüpfen zwei boolsche Werte zu einem boolschen Ergebnis. Die visuelle Programmiersprache unterstützt die gängigen boolschen Operationen und Vergleiche.

### 3.3.6 Kontrollstrukturen

Die visuelle Programmiersprache kennt verschiedene Kontrollstrukturen. Ihr sind *if-then-else*-Bedingungen, *while*- und *for*-Schleifen bekannt (siehe Bild 3.9 auf Seite 48). Die Sprache kennt auch Prozeduren ohne Parameter. Parameter hätten die Verwendung von lokalen Variablen mit sich gebracht, die gegenüber den globalen Variablen abgegrenzt werden müssen. Zum einem sind die Unterschiede zwischen lokalen und globalen Variablen und Parametern nicht einfach zu verstehen. Zum anderen ist auch die Realisierung aufwendig und wurde für spätere Versionen der Programmierumgebung aufgehoben.

Des weiteren gibt es noch Threads. Diese lassen sich deklarieren und starten. Ein Stoppen eines Threads von außerhalb ist in LeJOS nicht vorgesehen und wurde des-

halb nicht realisiert. Aus der Erfahrung heraus ist das externe Stoppen eines Threads immer eine heikle Angelegenheit und häufig damit verbunden, dass Datenstrukturen in einem ungünstigen Zustand hinterlassen werden, oder dass Deadlocks auftreten.

Ein weiteres mächtiges Konstrukt sind Events. Hiermit lassen sich Anweisungen nach Auftreten eines Ereignisses ausführen. Als Ereignisquellen stehen die Sensoren zur Verfügung. Die Ereignisbehandlung muss deklariert werden und kann bei Bedarf gestartet werden. Ein Stoppen der Ereignisbehandlung ist in LeJOS nicht vorgesehen.

## 3.4 Entwurf des Editors

Wie schon bei der Implementierung der beiden Prototypen zu sehen war, bot der von DEVIL erzeugte Editor schon die benötigte Funktionalität bezüglich Einfügen, Löschen und Drag&Drop. Die Handhabung ist etwas anders als unter Windows im allgemeinen erwartet wird. Eine Änderung des Verhaltens zieht jedoch tiefgreifende Veränderungen im Editor nach sich, die mit jeder neuen Version DEVILs einer Überarbeitung bedürfen.

Es gilt jedoch noch festzulegen, wie die Toolbar am linken Rand des Programm-Fensters aufgebaut ist. Es bietet sich eine Gruppierung der Piktogramme an, damit die Toolbar nicht überflutet wird. Die Gruppe wird durch ein Piktogramm repräsentiert, welches die in der Gruppe vorhandene Funktionalität darstellt. Es wurden folgende Gruppen festgelegt:

**Ausgaben** Hier sind die Funktionen zur Kontrolle der Motoren, Lampen und des LCDs untergebracht. Auch alle Modifizierer für die Ausgänge und die Leistung sind in dieser Gruppe zu finden. Die Numerischen Konstanten für die LCD-Ausgabe sind unter den Variablen zu finden.

**Zeit** Unter diesem Menüpunkt findet sich alle Funktionen, die sich mit der Zeit beschäftigen. Hier befinden sich Funktionen zum Abwarten eines Zeitraums und Funktionen zur Steuerung von Timern.

**Sensoren** Mit diesem Menüpunkt werden sämtliche Abfragemöglichkeiten von Sensoren zusammengefasst, sowie die dazu benötigten Parameter.

**Variablen** Die Gruppe *Variablen* umfasst das Erzeugen, Setzen und Abfragen von Ausdrücken sowie die Operationen zum Berechnen von ganzzahligen und boolschen Ausdrücken.

**Strukturen** Sämtliche Kontrollstrukturen wie Bedingungen, Schleifen, Prozeduren, Threads und Events. Die Events könnten auch auf die Sensoren- und Zeit-Gruppe verteilt werden, jedoch ist der Autor der Meinung, dass Events als mächtige Kontrollstruktur besser bei den Strukturen aufgehoben sind.

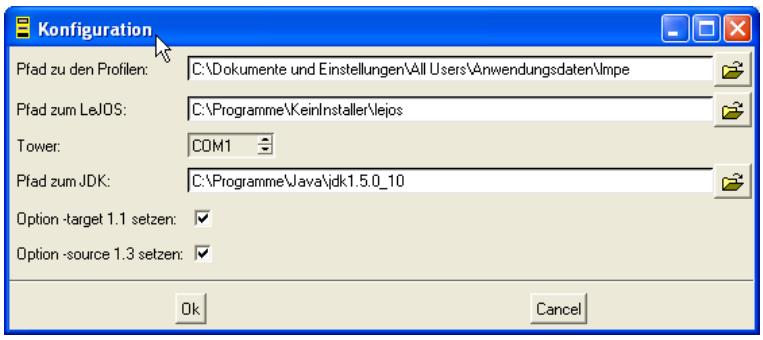


Abbildung 3.10: Die Konfiguration

Die Gruppen werden in einer Spalte der Toolbar angezeigt, eine mehrspaltige Toolbar ist auf Grund der geringen Menge nicht notwendig. Die Untermenüs der Toolbars könnten schon eine mehrspaltige Darstellung vertragen, jedoch bietet DEVIL diese Funktionalität nicht an.

## 3.5 Konfiguration

Der Editor benötigt für die Profile und für die Benutzung von LeJOS dauerhaft gespeicherte Konfigurationsdaten. Da die Profile in Kapitel 3.6 behandelt werden und dort auch die Einzelheiten der Konfiguration der Profile ausführlich behandelt werden, soll hier nur kurz erwähnt werden, dass die Konfiguration nur den Pfad zu der Profilkonfiguration enthält. Die Einzelheiten hierzu finden Sie in Kapitel 3.6.

Wie schon in Kapitel 2.4 beschrieben, wird zur Kompilierung und Übertragung des Programms ein JDK benötigt. Von dem JDK müssen die Java-Virtual-Machine und der Kompiler im Suchpfad liegen. Damit die Konfiguration vereinfacht wird, übernimmt der Editor das Eintragen des JDKs in den Suchpfad. Dabei handelt es sich aber nur um eine lokale Änderung des Suchpfades. Das heißt, dass nur das aufgerufene LeJOS-Skript den veränderten Suchpfad erbt, wohingegen das restliche System unberührt bleibt. Somit ist es möglich für die Programmierumgebung ein anderes JDK zu benutzen als für das restliche System. Der Pfad zum JDK soll durch ein Textfeld und einen Verzeichnis-Browser editierbar sein.

Der Pfad zum JDK wird auf Gültigkeit überprüft. Wenn `JDKPATH` der Pfad zum JDK ist, dann müssen unter `JDKPATH\bin\java.exe` und `JDKPATH\bin\javac.exe` die Java-Virtual-Machine und der Java-Kompiler erreichbar sein. Hinreichend ist es, dass diese beiden Dateien existieren. Ein Aufrufen der beiden Java-Tools und die anschließende Interpretation der Ausgaben würde die Komplexität unnötig erhöhen und die Zuverlässigkeit nur in Ausnahmefällen verbessern.

Je nach JDK-Version müssen dann noch die Optionen `--target` und `--source` entsprechend gesetzt werden (siehe Kapitel 2.4). Um die Implementierung einfach zu halten, wird auf eine automatische Feststellung der JDK-Version verzichtet. Statt dessen wird für jede Option ein Ankreuzfeld verwendet, wobei die Kreuze vom Ad-

| Windows | Unix |
|---|---|
| COM1 | com1 |
| COM2 | com2 |
| usb | /dev/usb/legousbtower0 |

Tabelle 3.1: Schnittstelle des Towers nach Plattform

ministrator entsprechend gesetzt werden müssen.

LeJOS benötigt die Umgebungsvariable `LEJOS_HOME` um die eigenen Bibliotheken zu finden. `LEJOS_HOME` ist der Pfad zum LeJOS-Hauptverzeichnis und wird ebenfalls von der Programmierumgebung benötigt um die Skripte von LeJOS aufzurufen. Wie auch der Pfad zum JDK wird der Pfad zu LeJOS durch ein Textfeld und einen Verzeichnis-Browser editiert. Zur Überprüfung werden die Skripte herangezogen. Wenn `LEJOS_HOME` der Pfad zu LeJOS ist, dann müssen diese Dateien vorhanden sein:

- `LEJOS_HOME\bin\firmdl.bat`
- `LEJOS_HOME\bin\lejosjc.bat`
- `LEJOS_HOME\bin\lejoslink.bat`
- `LEJOS_HOME\bin\lejosdl.bat`

Die Umgebungsvariable `LEJOS_HOME` wird nur für den entsprechenden Skriptaufruf gesetzt und hat keine Auswirkungen auf des restliche System.

Jetzt wird noch die Schnittstelle benötigt, an der der LEGO Tower angeschlossen ist. Hierbei muss zwischen den Betriebssystem-Varianten unterschieden werden. Eine Übersicht der verwendeten Schnittstellen je nach verwendeter Plattform findet sich in Tabelle 3.1. Zur Auswahl wird eine *SpinBox* verwendet, bei der sich der Administrator durch die einzelnen Möglichkeiten klicken kann. Eine Überprüfung der Eingabe ist nicht notwendig, da nur Werte aus Tabelle 3.1 eingegeben werden können. Somit können nur prinzipiell gültige Eingaben getätigt werden. Es kann jedoch nicht überprüft werden, ob sich hinter dieser Schnittstelle wirklich der Tower verbirgt. Dies kann nur durch eine Übertragung eines Programms zum RCX festgestellt werden. Die Schnittstelle des Towers wird in der Umgebungsvariablen `RCXTTY` abgelegt, die sich nur auf das aufgerufene Skript auswirkt.

Damit ist nur geklärt, was in der Konfiguration abgelegt wird. Nun muss überlegt werden, wo die Konfiguration abgespeichert wird. Bei diesen Konfigurationsdaten handelt es sich nicht um Benutzer-spezifische Daten, sondern um Rechner-spezifische Konfigurationsdaten. Deshalb muss die Konfigurationsdatei in einem entsprechenden Bereich abgelegt werden. Da auch die Daten der Profilkonfiguration noch in dem gleichen Bereich abgelegt werden soll, werden die Konfigurationsdateien in einem gemeinsamen Verzeichnis abgelegt.

Der Konfigurationsdialog soll im Menü untergebracht werden. Da es mit der derzeitigen Version von Devil nicht möglich ist ausserhalb des *Verarbeiten*-Menüs neue Menüeinträge zu erzeugen, kann der Menüeintrag für die Konfiguration nur im *Verarbeiten*-Menü erzeugt werden.

## 3.6 Profile

Robolab verfügt über ein *Level*-System um die grafische Programmiersprache in unterschiedlichen Komplexitätsstufen anzubieten (siehe Kapitel 2.3). Dabei gilt, dass je höher der *Level* ist umso mehr Sprachelemente dem Anwender angeboten werden. Mit diesem *Level*-System wird verhindert, dass die Schüler mit zu vielen Piktogrammen, Funktionen, Optionen und Sprachelementen überfordert werden. Somit werden nur die für den derzeitigen Lernfortschritt benötigten Sprachelemente angezeigt und nicht benötigte Menüs verborgen.

Leider sind die *Level* festgelegt und lassen sich nicht ändern um sie dem Lehrplan entsprechend anzupassen. In der Programmierumgebung soll dies geändert werden. Auch bekommen die *Level* jetzt einen neuen Namen, sie heißen jetzt *Profile*. Die *Profile* können nach Belieben konfiguriert werden. Auch ist die Anzahl der *Profile* beliebig wählbar. Leider gibt es auch eine Einschränkung. Die *Profile* wirken nur auf die grafische Programmiersprache und nicht auf die Menüs. Unter Devil ist es nur unter großen Schwierigkeiten möglich, Zugriff auf die Menüs und Kontextmenüs zu erlangen, so dass dies den Zeitrahmen dieser Arbeit gesprengt hätte.

Die Konfiguration der *Profile* soll eine hierarchische Darstellung anbieten, damit die Übersicht erhalten bleibt. Auf der obersten Ebene sind die Profile zu sehen. Die zweite Ebene bilden die Gruppen, die miteinander verwandte Sprachelemente zusammenfassen. Die letzte Ebene wird von den Sprachelementen selbst gebildet. Die Gruppierung der Sprachelemente orientiert sich an der Gruppierung der Sprachelemente in der Toolbar wie in Kapitel 3.3 beschrieben, wodurch sich der Lehrer als Betreuer der Profile besser orientieren kann.

Damit die Schüler die *Profile* nicht manipulieren können, muss der Zugriff auf die Konfigurationsdatei der *Profile* eingeschränkt werden. Das könnte zum einem von der Programmierumgebung durch einen Passwortschutz geleistet werden. Da heute in den Schulen Betriebssysteme eingesetzt werden, die eine Zugriffsbeschränkung auf Dateiebene unterstützen, lässt sich der Zugriff besser und für den Benutzer bequemer hierüber einschränken als das mit einer selbst entwickelten Zugriffsbeschränkung möglich wäre.

Bei Robolab kann der *Level* bei Programmstart ausgewählt werden. Dies wird auch bei LMPE so sein. Ein Ändern des Profils im laufenden Programm ist nicht so einfach zu realisieren. Dazu müsste Zugriff auf die internen Strukturen erlangt werden um den Zustand der Toolbar im laufenden Betrieb zu aktualisieren. Dies würde die Abhängigkeit von der internen Repräsentation in Devil immens erhöhen und es nötig machen bei jeder Version die Routinen zu überprüfen. Deshalb wird das Ändern des Profils im laufenden Betrieb verschoben.

Die Konfiguration der Profile ermöglicht natürlich das Erzeugen neuer Profile, sowie das Umbenennen und Löschen von existierenden Profilen. Da jedoch für den Fall, dass kein Profil konfiguriert ist, ein Standard-Profil vorhanden sein muss, wurde ein Standard-Profil programmatisch erzeugt. Dieses Standardprofil aktiviert alle Sprachelemente und kann weder gelöscht noch umbenannt werden. Da es programmatisch erzeugt wird, taucht das Standardprofil in keiner Konfigurationsdatei auf und ist somit auch bei Fehlen und nicht Lesbarkeit der Konfigurationsdatei vorhanden.

Da es schon ein Profil gibt, das immer existiert, wurde das Erzeugen neuer Profile durch das Kopieren ersetzt. Hierdurch kann gleichzeitig eine Art Prototyping implementiert werden in dem man ein existierendes Profil kopiert und dann seinen eigenen Wünschen entsprechend modifiziert. Dies hat einen ähnlichen Effekt wie Vererbung, ist jedoch wesentlich einfacher zu implementieren. Beim Kopieren muss überprüft werden, ob das Zielprofil nicht schon existiert. Das gleiche gilt für das Umbenennen.

# 4 Implementierung

In diesem Kapitel möchte ich die Besonderheiten der Implementierung beschreiben. Hier finden besonders die Vorgänge Erwähnung, die im Verborgenen des Editors stattfinden.

## 4.1 Grammatik und Visualisierung

Die Grammatik wurde entsprechend der Beschreibung in Kapitel 3.3 ab Seite 43 implementiert. Die meisten Klassen ergeben sich automatisch aus der Sprachbeschreibung. Einige Klassen sind jedoch dazugekommen, um die Programmierung zu vereinfachen oder gewisse Visualisierungen erst zu ermöglichen.

Ein Programm der visuellen Programmierumgebung besteht, wie in Kapitel 3.3 auf Seite 43 beschrieben, aus dem Körper und einer Deklarationsliste für Variablen, Zähler, Prozeduren Threads und Ereignisse. Daneben gibt es noch für jeden Sensor einen Eintrag. Diese Einträge dienen der korrekten Verwendung eines Eingang mit immer dem gleichen Sensortyp. So soll gewährleistet werden, dass nicht ein Eingang einmal als Berührungssensor und einmal als Lichtsensor verwendet wird. Ein Umtausch des Sensors im Betrieb ist unwahrscheinlich. Eine Erklärung des Sensortests findet sich bei der Beschreibung der Sensoren.

### 4.1.1 Icon

Das stärkste Beispiel zur Vereinfachung der Programmierung ist die abstrakte Klasse `icon`. Diese Klasse kapselt das Piktogramm eines Sprachelements und ist gleichzeitig auch die Elternklasse aller piktogrammbasierten Sprachelemente. Die Strukturbeschreibung sieht so aus:

```
ABSTRACT CLASS Icon {
  icon : VAL VLBoolean EDITWITH "None";
}
```

Die Visualisierung dazu hat folgenden Quelltext:

```
SYMBOL programView_Icon
INHERITS VPForm
COMPUTE
  SYNT.drawing = ADDROF(StatementDrawing);
END;
```

```
SYMBOL programView_Icon_icon
INHERITS VPFormElement, VPImagePrimitive
COMPUTE
  SYNT.formElementName = "icon";
END;
```

Die Icon-Klasse legt fest, dass alle Piktogramme im Editor dem Formularmuster folgen. Es wird als Defaultwert eine generelle Zeichnung mit nur einem Element, dem Piktogramm, festgelegt. Zur Verankerung des Piktogramms war es notwendig, das Attribut `icon` einzuführen. An diesem Attribut wurde das Piktogramm als Element `icon` des Formulars festgemacht, das Attribut wird nicht zur Speicherung eines Wertes verwendet. Der Name der generellen Zeichnung wird in der Regel in der spezialisierenden Klasse überschrieben und das Piktogramm wird erst dort gesetzt.

Eine Klasse, die von `Icon` erbt, erbt somit auch die Eigenschaften der visuellen Darstellung von `Icon`. Wenn etwas an der generellen Darstellung von Piktogrammen geändert werden soll, muss nicht jedes Piktogramm angefasst werden, sondern die Änderungen müssen nur in der Klasse `Icon` geändert werden. Trotzdem können weitere Attribute der erbenden Klasse einfach hinzugefügt werden, indem eine weitere generelle Zeichnung mit zusätzlichen Platzhaltern für die neuen Attribute erstellt wird. Mit dieser Zeichnung wird dann `Icon.drawing` überschrieben, und für die Attribute werden entsprechende Formular-Elemente in der Visualisierung beschrieben.

Fast alle Sprachelemente erben indirekt über die Klassen `Statement`, `Integer`, `Boolean` oder `Declaration` oder direkt von `Icon`. Es gibt noch weitere Klassen, die auf diese Weise visuelle Eigenschaften weitergeben, wie zum Beispiel die Klasse `StatementPortOutList`. Diese Klasse hält sowohl in dem Attribut `ports` eine Liste der verwendeten Ausgänge als auch deren Visualisierung. Wie der Name schon sagt, erbt `StatementPortOutList` auch von `Statement`, wodurch diese Klasse als Statement markiert wird. Diese Markerklasse ermöglicht es, sämtliche Anweisungen unter einem Namen zusammenzufassen. Genauso wurde mit den Gruppen Integer-Ausdrücke (`Integer`), boolsche Ausrücke (`Boolean`) und Deklarationen (`Declaration`) verfahren.

### 4.1.2 Ausgänge

Sämtliche Sprachelemente, die mit den drei Ausgängen zu tun haben, erben von `StatementPortOutList`. Die Ausgänge sollen sich in einer Liste unterhalb des Programms befinden. Jeder Ausgang darf nur höchstens einmal in dieser Liste verwendet werden. Deshalb wurde eine Synchronisationsprozedur geschrieben, die nach jeder Änderung der Liste der Ausgänge diese durchsucht und Doubletten entfernt. Am einfachsten war das nur einmalige Vorkommen eines der Ausgänge durch einen Sortieralgorithmus zu implementieren. Wenn ein Sortieralgorithmus eine Liste mit Doubletten sortiert, wird der Algorithmus irgendwann in die Verlegenheit kommen, die doppelten Elemente der Liste zu vergleichen. Bei dieser Gelegenheit kann eines

der doppelten Listenelemente entfernt werden. Als Nebeneffekt wird die Liste sortiert. Die Liste der Ausgänge kann höchstens nur vier Elemente enthalten: die drei Ausgänge A, B und C und einen doppelten Ausgang, der gerade hinzugefügt worden ist. Auf Grund dieser geringen Anzahl an Elementen brauchte der Sortieralgorithmus nicht effizient zu sein, deshalb wurde ein Bubble-Sort verwendet.

Zusätzlich wurde für die Liste der Ausgänge noch eine weitere Konsistenzbedingung definiert. Zwar darf die Liste der Ausgänge während des Editierens leer werden, jedoch muss zum Zeitpunkt der Kompilierung mindestens ein Ausgang in der Liste sein. Dies musste durch eine Konsistenzbedingung implementiert werden, da Devil die Anzahl der Elemente in einer Liste in dieser Weise nicht kontrollieren kann. Devil kann nur sicherstellen, dass entweder beliebig viele Elemente in einer Liste sind, höchstens ein Element in der Liste ist oder immer genau ein Element in der Liste ist.

Für die Geschwindigkeit wurde nur eine Klasse erstellt, die aber abhängig von der gespeicherten Geschwindigkeit ein entsprechendes Piktogramm anzeigt. Jeder Geschwindigkeit wurde ein entsprechendes Piktogramm zugeordnet. Dies wurde durch die Verwendung von Bedingungen in Lido erreicht. Bei der Codegenerierung konnte der Wert des Attributs dann direkt eingefügt werden. Dieses Verfahren wurde auch bei den Klassen `PortOut`, `PortIn` und bei den Operatoren angewendet. Hier der Quelltext für die Struktur und die Visualisierung:

```
CLASS Power INHERITS IntegerIcon {
  value : VAL VLInt INIT "8" EDITWITH
   "Scale -from 1 -to 8 -length 100";
}

SYMBOL programView_Power
COMPUTE
  SYNT.drawing = ADDROF(ConstantDrawing);
END;

SYMBOL programView_Power_icon
COMPUTE
  SYNT.imageName = IF(EQ(INCLUDING programView_Power.pers_value, 1),
    "img::power1",
    IF(EQ(INCLUDING programView_Power.pers_value, 2),
      "img::power2",
      IF(EQ(INCLUDING programView_Power.pers_value, 3),
        "img::power3",
        IF(EQ(INCLUDING programView_Power.pers_value, 4),
          "img::power4",
          IF(EQ(INCLUDING programView_Power.pers_value, 5),
            "img::power5",
            IF(EQ(INCLUDING programView_Power.pers_value, 6),
```

```
                "img::power6",
                IF(EQ(INCLUDING programView_Power.pers_value, 7),
                  "img::power7",
                  IF(EQ(INCLUDING programView_Power.pers_value, 8),
                    "img::power8",
                    "img::default" /* Falls falscher Wert kommt. */
                  )
                )
              )
            )
          )
        )
      )
    );
END;
```

Da DEVIL kein Äquivalent zu einer switch-Anweisung kennt, musste hier auf eine Kette von if-else-Bedingungen zurückgegriffen werden. Das Attribut `value` wurde durch einen Dialog editierbar gemacht, so dass die Geschwindigkeit durch einfaches Editieren des Wertes geändert werden kann. Damit der Wert nicht den Wertebereich von 1 bis 8 überschreitet, bietet der Dialog einen Schieberegler für den angegebenen Bereich an.

### 4.1.3 Eingänge

An jedem Eingang des RCX kann ein beliebiger Sensor angeschlossen werden. Da es unwahrscheinlich ist, dass sich während der Laufzeit eines Programms der Sensortyp eines Eingangs ändert, also zum Beispiel ein Berührungssensor durch einen Lichtsensor ausgetauscht wird, kann davon ausgegangen werden, dass während der Laufzeit eines Programms immer der gleiche Sensor an einem Eingang angeschlossen ist. Es muss daher verhindert werden, dass der Programmierer einen Eingang einmal als Berührungssensor abfragt und einmal als Lichtsensor.

Darüber hinaus gibt es auch bei der Kodegenerierung Probleme. Die Sensoren müssen initialisiert werden, wozu vor allem der Lichtsensor eine gewisse Zeit benötigt, damit er stabile Ergebnisse liefert. Deshalb findet die Initialisierung der Sensoren beim Programmstart statt, wodurch die Zwangspause durch die Initialisierung nicht störend auffällt. Da im Programm keine weitere Initialisierung der Eingänge stattfindet, muss der Sensortyp für jeden Eingang über die Laufzeit des Programms konstant sein.

Zur Lösung des Problems wurde der Root-Klasse für jeden Eingang ein `PortIn`-Objekt angelegt. Wird ein Eingang verwendet, so wird dieser referenziert. Jedes `PortIn`-Objekt enthält ein Attribut für den Namen, die Portbezeichnung (S1, S2 oder S3) und den Sensortyp. Der Sensortyp wird bei der Erzeugung des Objektes

auf *none*, also unbekannt gesetzt. Diese Initialisierung wird während der Erzeugung des Root-Objektes durch die Prozedur `edithook::create_Root` erledigt.

Bei jeder Referenzierung des Eingang wird der Sensortyp überprüft. In der von Devil angezeigten Auswahlliste werden nur die Eingänge angezeigt, die entweder noch nicht verwendet wurden, bei denen also der Sensortyp unbekannt ist, oder die den gleichen Sensortyp haben wie das zu referenzierende Objekt. Somit ist es nicht mehr möglich, einen Eingang mit zwei unterschiedlichen Sensortypen zu benutzen. Dieser Filter befindet sich in `checkutil::checkPortInType`.

```
proc checkutil::checkPortInType {tree value} {
  set type [getTypeOfObject [c::get $tree.PARENT.PARENT.PARENT]]
  if {$type eq ""} {
   return "Unbekannte Klasse, siehe Log!"
  }
  set portType [c::get "$value.type.VALUE"]
  if {$portType ne $type && $portType ne "none"} {
   # Entsprechend des Port-Typen eine Fehlermeldung ausgeben.
   switch -exact $portType {
   light {return "Dieser Port ist schon für Lichtsensoren \
   konfiguriert!"}
   touch {return "Dieser Port ist schon für \
   Berührungssensoren konfiguriert!"}
   }
  }
}
```

Die Prozedur `checkutil::checkPortInType` ist nicht für das Setzen des Sensortyps zuständig. Dies erledigt `synchook::PortIn`. Diese Prozedur erstellt eine Liste der Objekte, die einen bestimmten Eingang benutzen. Wenn diese Liste leer ist, dann wird der Sensortyp des Eingangs auf unbekannt (`none`) gesetzt. Dies ist der Fall, wenn der letzte Benutzer dieses Eingangs gelöscht worden ist. Wenn die Benutzerliste des Eingangs nicht leer ist, dann wird der Sensortyp des ersten Elements der Benutzerliste als Sensortyp für den Eingang verwendet. Eine weitere Untersuchung der Liste ist nicht notwendig, da immer nur ein weiterer Benutzer der Liste durch Referenzierung hinzugefügt werden kann. Der erste Benutzer legt den Sensortyp fest und alle weiteren Benutzer können auf Grund des Filters aus dem vorherigen Abschnitts nur den gleichen Sensortyp haben.

```
proc synchook::PortIn {portIn} {
set refererList [c::getList "$portIn.IVALUE.PARENT.PARENT.PARENT"]
if {[llength $refererList] == 0} {
set type "none"
} else {
set type [checkutil::getTypeOfObject [lindex $refererList 0]]
```

```
}
set portType [c::getList "$portIn.type.VALUE"]
if {$type ne $portType} {
edit::setAttr $portIn type $type
}
}
```

## 4.2 Kodegenerierung

Die Kodegenerierung verwendet IPTG, um aus Quelltext-Schnipsel Java-Quelltext für LeJOS zu erzeugen. Die Erzeugung wird mit Lido gesteuert. Der größte Teil der Kodegenerierung besteht darin, an einem Symbol die zur Kodegenerierung benötigten Daten aus den Attributen zu extrahieren und damit ein PTG-Objekt zu erzeugen. Diese PTG-Objekte werden dann zu Anweisunglisten oder Ausdrücken zusammengefasst.

Um die Körper von Schleifen, Prozeduren, Ereignissen, Threads und dem Programm selbst verarbeiten zu können, werden die Anweisungen in dem Körper mit einem `Constituents`-Befehl eingesammelt und mit `PTGNextStatement` zu einer Anweisungsliste verknüpft. Dabei gilt es jedoch etwas zu beachten:

Betrachten wir eine einfache, zählende Schleife. Sie beinhaltet neben einem Ausdruck für die Anzahl der Durchläufe noch den Namen der Zählvariable und einen Körper, der die Anweisungen der Schleife enthält. Der Ausdruck für die Anzahl der Schleifendurchläufe wird von der Klasse `StatementInteger` geerbt.

```
CLASS For INHERITS StatementInteger {
  name : VAL VLString INIT "Zähler";
  body : SUB Body%;
}

ABSTRACT CLASS StatementInteger INHERITS Statement {
  expression : SUB Integer!;
}
```

Das Ziel ist es, folgende IPTG-Klasse zu füllen:

```
For(name, expression, body):
  for (int [name] = 1; [name] <= [expression]; [name]++) {
    [body]
  }
```

Der Körper `body` enthält die Anweisungsliste der Schleife. Wie zu sehen ist, enthält damit die Schleife schon alle Anweisungen des Körpers. Wenn mehrere Schleifen ineinander verschachtelt werden, existiert mindestens ein `PTGFor` unterhalb des betrachteten Schleifenkörper. Ein einfaches Constituents würde jetzt auf das `For` treffen, `PTGFor` samt enthaltenen Schleifenköper zur Anweisungsliste hinzufügen und

danach in den Körper von `For` hinabsteigen. Somit werden auch die Anweisungen des Körpers der Anweisungsliste hinzugefügt und sind doppelt enthalten. Um dieses zu verhindern, können betreffende Symbole mit einer `SHIELD`-Klausel geschützt werden. Um hier nicht all zu viele Einträge zu erhalten und die Anzahl der zu modifizierenden Stellen im Quelltext gering zu halten, wurde der Schleifenköper in der Klasse `body` gekapselt. Somit muss der Körper innerhalb der Schleife nur mit einem einfachen `CONSTITUENT` ermittelt werden, was gegen die Anweisungen in `Body.statements` geschützt ist. Das Constituents in dem Symbol Body wird gegen Body geschützt. Somit ist die Erzeugung des Schleifenkörpers von der Schleife unabhängig geworden und ein Hinzufügen oder Entfernen eines weiteren Schleifentyps erfordert keine Änderungen mehr.

Die Kodegenerierung des Motors war nicht trivial. Das Problem ist, dass am Motor die Ports als Liste von bis zu drei Ausgängen und die Geschwindigkeit hängen. Diese beiden Parameter des Motors sind nicht als Attribute, sondern als Unterbäume (`SUB`) ausgeführt, um im Editor ein ähnliches Verhalten wie in ROBOBLAB zu erhalten. Somit ist ein direkter Zugriff vom Motor aus nicht mehr möglich, sondern die Daten müssen mit `CONSTITUENT` ermittelt werden.

Verschärfend kommt noch hinzu, dass die eigentlich Kodegenerierung nicht im Motor stattfinden kann, sondern in der Portliste muss der Quelltext generiert werden. Dies liegt darin begründet, dass LeJOS nur immer einen Motor ansteuern kann, die visuelle Programmiersprache jedoch mehrere Motoren auf ein Mal. Von der Portliste aus kann jedoch nicht auf die Geschwindigkeiten zugegriffen werden.

Gelöst wurde das Problem, indem die Geschwindigkeit in einem Attribut des Motors zwischengespeichert wurde. Dieses Attribut ist von der Portliste aus erreichbar und kann per `INCLUDING` eingebunden werden. Somit wird für jeden Port in der Portliste eine Befehlszeile erzeugt, die genau einen Motor erzeugt. Diese Befehlszeilen werden dann im Motor mit einem `CONSTITUENTS` eingesammelt.

## 4.3 Kompilierung und Download

Die Übersetzung des Java-Quelltextes und die Übertragung des Programms wurden schon im Prototypen fast vollständig implementiert. Somit mussten nur ein paar fehlende Ausnahmefälle beachtet werden und der Quelltext wurde noch mal zwecks einer besseren Strukturierung überarbeitet.

Nach Abschluss der Implementierungen für die Übersetzung und Übertragung zum RCX wurde eine neue Version von LeJOS veröffentlicht. Die darin enthaltenen Scripte verhalten sich bezüglich der Rückgabe von Fehlerwerten anders als in der vorherigen Version. Somit unterstützt die Programmierumgebung nur die ältere Version 2.1.0 von LeJOS.

| Schlüssel | Standardwert |
|---|---|
| lejos | "‘"’ |
| jdk | "‘"’ |
| source | "‘0"’ |
| target | "‘1"’ |
| profiles | "‘%ALLUSERPROFILE%\Anwendungsdaten\lmpe"’ |
| tty | "‘COM1"’ |

Tabelle 4.1: Datenstruktur der Konfiguration

## 4.4 Konfiguration

Die Konfiguration wird komplett in Tcl/Tk implementiert, da Devil keinerlei Unterstützung für die Anlage von Konfigurationsdaten bietet. Um wie in Kapitel 3.5 beschrieben den Eintrag im *Verarbeiten*-Menü anzulegen, muss ein Prozessor angelegt werden. Da der Konfigurationsdialog komplett in Tcl/Tk geschrieben wird, erfolgt auch die Registrierung des Prozessors bei Devil durch den Aufruf der Prozedur `file::registerProcessor`. Als Parameter werden der Bezeichner des Prozessors, die aufzurufende Prozedur und der im Menü darzustellende Text angegeben. Dabei ist der Bezeichner des Prozessors in unserem Fall nicht von Interesse und kann beliebig gewählt werden.

Die aufzurufende Prozedur muss einen Parameter haben, über den der Wurzelknoten des Syntaxbaumes an den Prozessor übergeben wird. Da wir jedoch in unserem Konfigurationsdialog keinen Zugriff auf den abstrakten Syntaxbaum benötigen, wird auch der Wurzelknoten des Syntaxbaumes in der Prozedur `config::configDialog` nicht verwendet.

Bevor wir uns weiter mit dem Konfigurationsdialog beschäftigen, wollen wir uns die benötigten Datenstrukturen genauer ansehen. Unter Tcl werden solche Strukturen auf assoziative Felder abgebildet. Die dabei verwendeten Schlüssel und Standard-Werte können der Tabelle 4.1 entnommen werden.

Um die Struktur zu verbergen wurden *Getter*- und *Setter*-Prozeduren für die einzelnen Einträge geschrieben. Somit wird sichergestellt, dass immer die gleichen Schlüssel zum Zugriff auf das Feld verwendet werden und eventuelle Änderungen der Schlüssel nur an wenigen Stellen durchgeführt werden müssen. Zusätzlich wurden in den *Setter*-Prozeduren die in Kapitel 3.5 beschriebenen Plausibilitätstests implementiert. Somit ist an zentraler Stelle sichergestellt, dass nur korrekte Einträge in die Konfiguration gelangen.

Das Laden und Speichern der Konfiguration wurde mit den in Tcl vorhandenen Bordmitteln implementiert. Das Feld wird beim Speichern mittels `array get` in eine Liste umgewandelt, die dann in eine Datei geschrieben wird. Da Tcl Listen als Texte darstellt, sind die Konfigurationsdateien im Notfall vom Menschen lesbar und können editiert werden. Es muss jedoch gesagt werden, dass diese Listen mit zunehmender Größe sehr schnell unübersichtlich werden.

Das Laden der Konfiguration erfolgt in genau umgekehrter Reihenfolge. Die Datei wird eingelesen und eine Variable wird mit der Liste gefüllt. Diese Liste wird dann mit `array set` in ein Feld umgewandelt. Da die Konfiguration keine großen Datenmengen umfasst, kann auf ein blockweises Einlesen der Daten verzichtet werden.

Für die Benutzerschnittstelle werden hauptsächlich Dialoge verwendet, die Knöpfe zum Bestätigen oder Abbrechen der Konfiguration besitzen. Devil stellt mit `CDlg::createDialog` schon einen entsprechenden Rahmen zur Verfügung. Als Parameter werden unter anderem die Prozeduren für den *OK*- und den *Cancel*-Knopf übergeben. Diese Prozeduren werden beim Drücken des entsprechenden Knopfes aufgeführt. `CDlg::createDialog` gibt den Dialog zurück und es können weitere Widgets hinzugefügt werden.

Das Problem mit diesem Standard-Dialog von Devil ist, dass die Prozedur für den OK-Knopf keinen Zugriff auf die Widgets hat. Somit muss eine Datenübergabe mit globalen Variablen hergestellt werden, was ein schlechter Programmierstil ist. Deshalb wurde die Prozedur `CDlg::createDialog` kopiert und dahingehend verändert, dass den Prozeduren für den *OK*-Knopf und *Cancel*-Knopf als Parameter der Dialog übergeben wird. Dadurch ist es möglich auf alle Widgets des Dialoges zuzugreifen. Dieser neue Dialog ist unter `config::stdDialog` zu finden.

## 4.5 Profile

Die Profile mit ihrem hierarchischen Aufbau benötigen eine komplexe Struktur zu ihrer Speicherung. Da Tk jedoch keine Bäume oder ähnliches zur Verfügung stellt, muss die Struktur mit den vorhandenen Bordmitteln, also Listen und assoziativen Feldern, simuliert werden.

Wie schon in Kapitel 3.6 beschrieben, bilden die Profile einen Baum. Die oberste Ebene ist eine Liste von Profilnamen. Somit wird ein assoziatives Feld als globale Variable mit dem Namen `profiles` angelegt. Dort wird eine Liste mit Profilen unter dem Schlüssel `profile.list` angelegt. Die zweite Ebene wird durch die Gruppen repräsentiert. Also gibt es auch eine Liste von Gruppen unter dem Schlüssel `groupList`. Da in jedem Profil sämtliche Gruppen vertreten sind, existiert keine Liste, die die Gruppen einem bestimmten Profil zuordnet. In den Gruppen gibt es die Sprachelemente, auch *Labels* genannt. Da eine Gruppe nur bestimmte *Labels* enthält, gibt es für jede Gruppe eine Mitgliederliste unter dem Schlüssel `gruppe.groupMembers`. Hiermit ist der Profilbaum schon fertig und könnte dargestellt werden.

Es tritt allerdings ein Problem auf. Devil identifiziert die Sprachelemente über Labels, die bei der Verwendung von Piktogrammen gleich den Namen der Piktogramme sind. Diese folgen dem Format `img::DateinameOhneErweiterung`, was für den Anwender nicht unbedingt verständlich ist. Also wird noch eine Tabelle zur Übersetzung von *Label* in Klartext benötigt. Dafür wird für jedes Label unter dem Schlüssel `LABEL` der Klartext abgelegt. Da die Gruppen von den Menüs in der Toolbar abgeleitet sind, werden dort auch *Label* verwendet, somit müssen auch für Gruppen

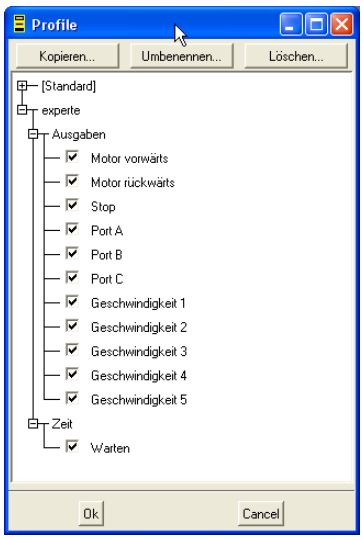


Abbildung 4.1: Der Konfigurationsdialog für die Profile

Klartexte abgelegt werden unter dem Schlüssel `GRUPPE`.

Nun muss noch für jedes Sprachelement in jedem Profil abgelegt werden, ob es aktiviert ist. Dazu wird für jedes Sprachelement ein Eintrag mit dem Schlüssel `profile.PROFIL.LABEL` abgelegt. Wenn das Element aktiviert ist, muss in diesem Eintrag eine `1` stehen, sonst eine `0`.

Hiermit wäre die komplette Datenstruktur der Profile beschrieben. Mehr wird nicht benötigt. Es sei noch darauf hingewiesen, dass die Strukturen für Gruppen und Sprachelemente programmatisch erzeugt werden, da sie von der verwendeten Grammatik und somit von der Programmierumgebung abhängen. Die Gruppen und Sprachelemente werden deshalb auch nicht abgespeichert.

Wir wollen uns jetzt den Grundoperationen zuwenden. Um ein Profil zu erzeugen muss der Name des Profils der Profilliste `profile.list` hinzugefügt werden. Danach wird für jedes Sprachelement ein Eintrag mit dem Schlüssel `profile.PROFIL.LABEL` und Wert `1`, also aktiviert, erzeugt. Das Löschen funktioniert analog umgekehrt. Das Profil wird aus der Profilliste entfernt und die Einträge für Label werden ebenfalls gelöscht.

Der Konfigurationsdialog arbeitet nicht direkt auf diesen Strukturen. Er liest die Struktur einmal aus um die Baumkomponente zu füllen. Danach werden alle Arbeiten innerhalb der Baumkomponente vollzogen. Die Änderungen werden erst übernommen, wenn der Konfigurationsdialog mit dem `OK`-Knopf beendet wird.

Da das Tk über keine Baumkomponente verfügt, musste eine Bibliothek gefunden werden, die eine Baumkomponente zur Verfügung stellt. Dabei kann zwischen Bibliotheken gewählt werden, die komplett in Tcl implementiert sind, oder Binärbibliotheken, die in C/C++ implementiert sind. Die Binärbibliotheken sind sehr schnell, haben jedoch den Nachteil, dass man für jedes Betriebssystem eigene Versionen der Bibliotheken benötigt. Tcl-Bibliotheken sind nicht ganz so schnell, da

sie, wie jeder anderer Tcl-Code auch, interpretiert werden. Dafür läuft die gleiche Bibliothek auf allen Betriebssystemen. Bei den heutigen Rechenleistungen ist die Geschwindigkeit für die Darstellung einer Baumkomponente nahezu egal. Da die in unserem Fall darzustellenden Datenmengen gering sind, ergibt sich kein bemerkbarer Geschwindigkeitsunterschied. Es wurde sich für die in Tcl geschriebene Bibliothek *BWidgets* entschieden, da sie in Tcl geschrieben ist, keine Klassen- beziehungsweise Objekt-Erweiterung erfordert und ordentlich dokumentiert ist.

Die im Konfigurationsdialog möglichen Operationen, das Kopieren, Umbenennen und Löschen von Profilen wirken nur auf die Baumkomponente und werden erst bei Bestätigung des Dialogs übernommen. Die Implementation dieser Operationen ist Routine. Das Kopieren und Löschen wurde implementiert und beim Umbenennen das Profil kopiert und dann das alte Profil gelöscht. Beim Löschen und Umbenennen ist darauf zu achten, dass das Standardprofil nicht modifiziert wird und beim Kopieren und Umbenennen sollte der Name des neuen Profils noch nicht existieren damit der Name des Profils eindeutig bleibt.

Die Übernahme der Daten von der Baumkomponente in die interne Struktur geschieht nach dem Synchronisationsprinzip. Zuerst werden in der Baumkomponente nicht mehr vorhandene Profile aus der internen Struktur gelöscht. Danach werden in der internen Struktur noch nicht existierende Profile erzeugt. Jetzt existieren in der Baumkomponente die gleichen Profile wie in der internen Struktur und die Werte der Labels müssen nur noch herüberkopiert werden.

Abschließend müssen noch das Laden und Speichern beschrieben werden. Das Speichern ist sehr einfach gehalten. Dem Kommando `array get` kann als Parameter ein *Wildcard* mitgegeben werden, so dass nicht alle Elemente des Feldes in eine Liste umgewandelt werden, sondern nur die, die dem *Wildcard* entsprechen. Also müssen die Elemente des Feldes, die in die Konfigurationsdatei sollen, mit dem gleichen Wort beginnen. Ich habe das Wort `profile` gewählt, also sieht der Befehl so aus:

```
array get profiles "profile.*"
```

Danach wird die Liste wie schon in Kapitel 4.4 in die Konfigurationsdatei geschrieben.

Beim Laden könnte analog zu der Konfiguration das umgekehrte Verfahren angewendet werden, da `array set` nur die Einträge im Feld verändert, die in der als Parameter übergebenen Liste stehen. Die anderen Einträge des Feldes bleiben unberührt. Wie verhält man sich jedoch, wenn sich die Grammatik derartig verändert hat, dass Sprachelemente weggefallen sind, hinzugekommen oder umbenannt worden sind? Die eine Möglichkeit wäre, das Problem einfach zu ignorieren. Dann würden sich mit der Zeit immer mehr tote Sprachelemente ansammeln, die nicht einmal angezeigt werden, da sich der Algorithmus zum Befüllen der Baumkomponente immer an der Gruppenliste und der Mitgliederliste der Gruppen orientiert und diese der Grammatik des gerade verwendeten Editors entsprechen.

Um dieses zu vermeiden, wird die Profilkonfiguration in ein temporäres Feld geladen. Dann durchläuft die Programmierumgebung die ihr bekannten Sprachprofile

und kopiert sie in die interne Struktur. Sollte dabei ein Sprachprofil in dem temporären Feld nicht gefunden werden, muss es ein neues Sprachelement sein und es erscheint der Hinweis, dass die Profile überprüft werden müssen. Das neue Sprachelement wird daraufhin aktiviert. Gelöschte Sprachelemente, die der Editor nicht mehr kennt, aber die noch in der temporären Struktur enthalten sind, werden ignoriert.

Den Fall der umbenannten Sprachelemente deckt dieser Algorithmus nur unzureichend ab. Für den Algorithmus stellt sich ein umbenanntes Element als ein neues und ein gelöschtes Sprachelement dar. Das gelöschte Sprachelement mit dem alten Namen beinhaltet den Status und wird verworfen. Das neue Sprachelement mit dem neuen Namen wird wie oben beschrieben aktiviert. Somit geht der alte Zustand des Sprachelements verloren. Eine Lösung des Problems wurde verschoben, da

1. Umbenennungen von Sprachelementen selten auftreten. Die für die Konfiguration wichtigen Namen sind die *Labels*. Die *Labels* sind die Namen der Piktogramme der Toolbar, die wiederum unabhängig von den verwendeten Klartextnamen sind. Also kann in ziemlich weiten Grenzen auf eine Umbenennung des Labels verzichtet werden, da davon unabhängig der Klartextname geändert werden kann.

2. Der Aufwand für eine korrekte Implementierung ist sehr hoch und fehleranfällig. Es müsste eine Historie für die Umbennung der *Label* mit den dazugehörigen Versionsnummern erstellt werden und diese müsste auch noch lückenlos gepflegt werden. Diese Historie müsste in das Programm eingearbeitet werden, damit beim Laden des Profils die Umbenennungen durchgeführt werden können und zu den *Label* müsste auch immer die Versionsnummer abgelegt werden, damit die Umbenennungen auch in der richtigen Reihenfolge ausgeführt werden.

# 5 Zusammenfassung und Ausblick

Im großen und ganzen konnten die in der Aufgabenstellung gestellten Ziele erreicht werden. Es wurde eine visuelle Programmiersprache geschaffen, die ROBOLAB ähnlich ist wodurch ein einfacher Umstieg gewährleistet ist. Viele der guten Eigenschaften von ROBOLAB konnten übernommen werden. Die Programmiersprache kommt weiterhin ohne Kreuzungen der Verbinder aus. Bis auf die Anzahl der verwendeten Piktogramme ist die Anzahl der unterschiedlichen graphischen Elemente gering und somit leicht erlernbar. Die Anzahl der Piktogramme ließ sich nicht wesentlich reduzieren, da dann die Sprache komplizierter wird und der Lernaufwand steigt, bis ein Programmieranfänger ein funktionsfähiges Programm erstellen kann.

## 5.1 Visuelle Programmiersprache

Der Sprachumfang wurde gegenüber ROBOLAB reduziert. Viele spezielle Sprachelemente wurden nicht realisiert, da sie nur eine Spezialisierung eines allgemeineren Sprachelements darstellen. Somit wurde die Funktionalität der Sprache gewahrt, jedoch litt die Übersichtlichkeit. Das Warten auf die Veränderung einer Variablen muss nun beispielsweise mit einer Schleife, einer Bedingung und einer Variablen zum Ablegen des alten Wertes von Hand programmiert werden. So ein Konstrukt muss vom Leser des Programms erst erkannt werden.

Im Verlauf dieser Arbeit konnten nur globale Variablen implementiert werden. Eine Erweiterung um lokale Variablen wäre noch empfehlenswert. Damit einhergehend können die Prozeduren noch um Parameter erweitert werden. Beide Erweiterungen würden die Fähigkeiten der visuellen Programmiersprache zur strukturierten Programmierung verbessern.

Die Ereignisse scheinen im Vergleich zu Robolab ziemlich reduziert, jedoch ist dies in erste Linie dem Ereignismodell von LeJOS geschuldet. Eine Erweiterung um die Handhabung zu vereinfachen erscheint jedoch möglich. Zur Vervollständigung der Ereignisse fehlt auch noch die Möglichkeit, selbst Ereignisse definieren zu können.

Die Threads sind zwar funktionsfähig vorhanden, jedoch sind die Möglichkeiten zur Synchronisation beschränkt. Zur Zeit können sich Threads nur über die globalen Variablen synchronisieren und diese Art Synchronisation muss komplett von Hand implementiert werden. Auch gibt es noch keine Möglichkeit, Ressourcen durch Semaphore gegen den gleichzeitigen Zugriff zweier Threads zu schützen. Die Implementierung der Threads ist noch als rudimentär anzusehen und bietet noch Erweiterungspotential.

## 5.2 Profile und Konfiguration

Die Profile konnten gegenüber Robolab wesentlich verbessert werden und ermöglichen dem Lehrpersonal freie Konfigurierbarkeit der Profile sowie deren Namen und Anzahl. Dadurch lässt sich die Programmiersprache den Bedürfnissen des Unterrichts anpassen.

Die Profile wirken jedoch ausschließlich auf die Programmiersprache, den Editor beeinflussen sie gar nicht. Es wäre schön, wenn die Programmierumgebung noch so erweitert werden kann, dass über die Profile auch Einfluss auf die Menüs, Kontextmenüs und die Toolbar oberhalb des Fensterbereiches genommen werden kann. Hierzu müssen im Editor etliche Prozeduren überschrieben werden.

Seit kurzem unterstützt Devil die Möglichkeit, eigene Menüs außerhalb des Menüs für Prozessoren zu erzeugen. Vielleicht kann diese Funktionalität auch zum Zugriff auf die Standardmenüs verwendet werden.

Die Konfiguration wurde vollständig implementiert und alle benötigten Funktionen wurden umgesetzt. Für die Verwendung mit LeJOS sind keine Erweiterungen notwendig.

## 5.3 Kodegenerierung und Kompilierung

Die Kodegenerierung wurde vollständig implementiert und erzeugt Java-Quelltext für das LeJOS-System. Danach wird der Quelltext durch die JavaTools von LeJOS übersetzt, zu einer Datei gelinkt und zum RCX übertragen. Alle dazu nötigen Funktionen wurden implementiert und benötigen keine weitere Überarbeitung.

Vielleicht wäre eine Erweiterung der Kodegenerierung um NQC erwägenswert. Diese Sprache ist eine C-ähnliche Programmiersprache auf Basis der Firmware von LEGO. Sie hat den Vorteil, dass sie parallel mit Robolab oder RCX code verwendet werden kann, da keine neue Firmware aufgespielt werden muss. Deshalb steht zur Programmierung mehr Speicher zu Verfügung.

Gleichzeitig muss aber auch gesagt werden, dass NQC keine objektorientierte Programmiersprache ist und somit im Zusammenhang des hier vorausgesetzten Unterichtsszenarios nicht einsetzbar ist. Es würde aber die Einsetzbarkeit der Programmierumgebung stark erweitern.

## 5.4 Hardware

Während die Arbeit fertig gestellt wurde, brachte LEGO eine neue Version von Mindstorms heraus, die auf den NXT basiert. Der NXT ist eine Weiterentwicklung des RCX. Äußerlich bekommt der NXT nun ein ordentliches Display, das Grafiken und mehrzeiligen Text darstellen kann. Die Anzahl der Eingänge wurde auf vier erhöht, während die Anzahl der Ausgänge konstant geblieben ist. Es wurden weitere Sensoren entwickelt und die Anbindung an den Computer erfolgt jetzt entweder mit Kabel über USB oder drahtlos mit Bluetooth. Auch das Innere des NXT wurde

gegenüber RCX auf den aktuellen Stand der Technik gebracht und verfügt jetzt über ausreichend RAM und Rechenleistung.

Zur Zeit wird eine LeJOS-Version für den NXT entwickelt, jedoch steckt sie im Moment noch im Alpha-Stadium. Wenn LeJOS für NXT einen stabilen Stand erreicht hat, ist eine Einbindung in die Programmierumgebung sicherlich empfehlenswert.

# Literaturverzeichnis

[Appel 2002] Appel, Andrew W.: *Modern compiler implementation in Java.* second. Cmabridge University Press, 2002. – ISBN 0–521–82060-X

[DEVIL Benutzerhandbuch 2007] : *Suche.* Wikepedia. March 2007. – URL `http://ag-kastens.uni-paderborn.de/forschung/devil/documentation/manual-html.gen/ma`

[Grand 1998] Grand, Mark: *Patterns in Java: a catalog of reusable design patterns.* John Wiley & Sohns, Inc., 1998. – ISBN 0–471–25839-3

[Kastens u. a. 2006] Kastens, Uwe ; Waite, William M. ; Slone, Anthony M.: *Generating Software from Specifications.* Jones and Bartlett, 2006

[LeJOS 2002] : *README.* LeJOS 2.1.0. Dezember 2002. – URL `http://lejos.sourceforge.net/`

[Schiffer 1998] Schiffer, Stefan: *Visuelle Programmierung.* Addison-Wesley-Longman, 1998

[Stuber und Denham 2006] Stuber, Jürgen ; Denham, Tom: *Re: Garbage Collection.* Mailarchiv von LeJOS. April 2006. – URL `http://sourceforge.net/mailarchive/message.php?msg_id=15352981`. – Zugriffsdatum: 29.04.2006

[Visuelle Programmiersprache 2007] : *Suche.* Wikepedia. March 2007. – URL `http://de.wikipedia.org/`

[Watt und Brown 2000] Watt, David A. ; Brown, Deryck F.: *Programming Language Processors in Java.* Person Education Limited, 2000. – ISBN 0 130 25789 9

# Selbstständigkeitserklärung

Ich versichere, dass ich die Diplomarbeit zum Thema "Entwurf einer grafischen Entwicklungsumgebung zur Programmierung des LEGO Mindstorms RCX in Java" selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Alle Stellen die dem Wortlaut oder Sinn nach anderen Werken entnommen sind, habe ich unter Angabe der Quelle kenntlich gemacht.

Bielefeld, den 28.10.2006

Christian Weddeling